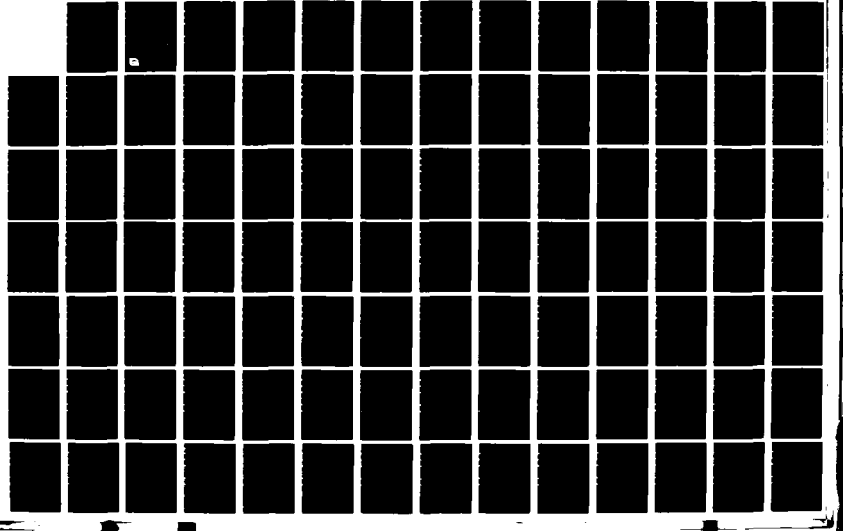
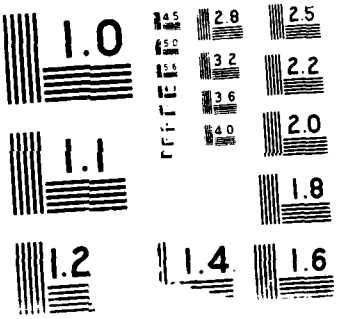
AD-A194 356 STRATEGIC DEFENSE INITIATIVE ARCHITECTURE DATAFLOW MODELING TECHNIQUE VER (U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA J L LINN ET AL. 22 AP 88 1/2

UNCLASSIFIED IDA-P-2035 IDA/HQ-87-32623 MDA903-84-C-0031 F/G 15/3.1 NL




1.0
1.1
1.2
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

IDA PAPER P-2035

STRATEGIC DEFENSE INITIATIVE ARCHITECTURE DATAFLOW MODELING TECHNIQUE VERSION 1.5

Joseph L. Linn
Cy D. Ardoin
Cathy Jo Linn
Stephen H. Edwards
Michael R. Kappel
John Salasin

April 1988

Prepared for
Strategic Defense Initiative Organization (SDIO)

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, or (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Papers

Papers normally address relatively restricted technical or policy issues. They communicate the results of special analyses, interim reports or phases of a task, ad hoc or quick reaction work. Papers are reviewed to ensure that they meet standards similar to those expected of refereed papers in professional journals.

Memorandum Reports

IDA Memorandum Reports are used for the convenience of the sponsors or the analysts to record substantive work done in quick reaction studies and major interactive technical support activities; to make available preliminary and tentative results of analyses or of working group and panel activities; to forward information that is essentially unanalyzed and unevaluated; or to make a record of conferences, meetings, or briefings, or of data developed in the course of an investigation. Review of Memorandum Reports is suited to their content and intended use.

The results of IDA work are also conveyed by briefings and informal memoranda to sponsors and others designated by the sponsors, when appropriate.

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

This paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and sound analytical methodology and that the conclusions stem from the methodology.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE AD-A194356

1a REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Public Release/Unlimited Distribution
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) IDA Paper P-2035	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses 6b OFFICE SYMBOL IDA	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION Strategic Defense Initiative Organization 8b OFFICE SYMBOL (if applicable) SDIO	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) BM/C3, Room 1E149 The Pentagon Washington, D.C. 20301-7100	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-R5-422 / WORK UNIT ACCESSION NO.
11 TITLE (Include Security Classification) Strategic Defense Initiative Architecure Dataflow Modeling Technique, Version 1.5 (U)	
12 PERSONAL AUTHOR(S) Joseph L. Linn, Cy D. Ardoin, Cathy Jo Linn, Stephen H. Edwards, Michael R. Kappel, John Salasin	
13a TYPE OF REPORT Final 13b TIME COVERED FROM _____ TO _____ 14 DATE OF REPORT (Year, Month, Day) April 22 1988 15 PAGE COUNT 159	
16 SUPPLEMENTARY NOTATION	

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Computer architecture; System architecture; Ada programming language; Battle Management (BM); Command, Control, and Communications (C3); Software tools and techniques; Modeling and simulation; Program design language; Process description language; Software engineering; Interfaces; Compilers; Evaluation.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

IDA Paper P-2035 presents the SDI Architecture Dataflow Modeling Technique (SADMT), a uniform formal notation for the description of SDI system architectures and Battle Management and Command, Control, and Communications (BM/C3) architectures. SADMT is a technique for thinking about and describing architectural processses and structures that use the typing and functional facilities of the Ada programming language. This document defines SADMT and the programming interface to the SADMT Simulation Facility (SADMT/SF). The issues addressed here are those relevant to providing formal descriptions of system structure and behavior for interface consistency checking, system simulation, and system evaluation.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ■ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Cathy Jo Linn	22b TELEPHONE (Include Area Code) (703) 824-5520 22c OFFICE SYMBOL IDA/CSED

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PA

IDA PAPER P-2035

STRATEGIC DEFENSE INITIATIVE ARCHITECTURE DATAFLOW MODELING TECHNIQUE VERSION 1.5

Joseph L. Linn
Cy D. Ardoin
Cathy Jo Linn
Stephen H. Edwards
Michael R. Kappel
John Salasin

April 1988

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-R5-422

Acknowledgments

Special acknowledgement is due to LTC David R. Audley for his vision and concern about the need for rigor in the SDI design process. This work would not have been possible without his support, encouragement, and insight. Similarly, the demonstration system and the greatly improved initial SADMT revision would have been impossible without the continuing support of LTC Jon Rindt and CAPT David A. Hart.

We are also very grateful to Prof. David C. Luckham and Goran Hemdal for their valuable technical advice, both the criticism and the encouragement. In addition, we thank the many reviewers who provided insight and assistance through the peer review process.

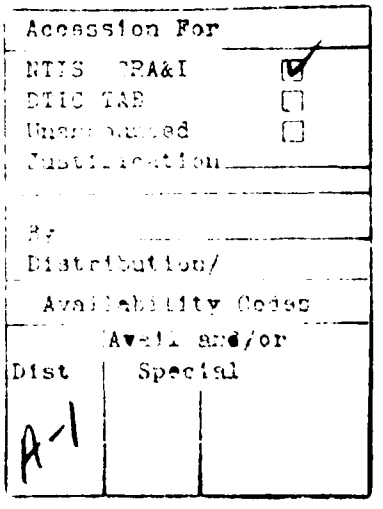


TABLE OF CONTENTS

CHAPTER 1

Introduction

This paper presents the first published revision of the first version of the SDI Architecture Dataflow Modeling Technique (SADMT). SADMT provides a uniform formal notation for the description of SDI system architectures and Battle Management and Command, Control, and Communications (BM/C^3) architectures. SADMT is a technique for thinking about and describing architectural processes and structures that uses the typing and functional facilities of the Ada programming language. However, SADMT is not a(n Ada) Program Design Language, particularly not in the sense in which this term is used in the Ada community[1]. Thus, when one uses SADMT, one must think in SADMT terms, *even though the final representation of SADMT is in terms of Ada!* Moreover, SADMT is not a replacement for the numerous other methods of describing architectures that have been used in past studies, but is an additional means of description. Such a uniform formal description is needed to facilitate:

a) evaluation and comparison of architectures and development of standard tools for the analysis of architectures,

b) standardization of architectural specification for simulation purposes, and

c) development of new architectures based on characteristics found to be desirable in previous architectural studies.

1.1. Scope

This document defines SADMT and the programming interface to the SADMT Simulation Framework (SADMT/SF). The issues addressed here are those relevant to providing formal descriptions of system structure and behavior for interface consistency checking, system simulation, and system evaluation. Related issues yet to be resolved but not addressed here include:

a) the use of structured "natural language" comments for documentation,

b) the control interface for the simulation environment, and

c) support for configuration management and version control.

1.2. Background

In a draft policy statement issued by General O'Neill on July 29, 1986, Ada was selected as the basis for a uniform formal Process Description Language (PDL).[2] Ada is a stable language; its definition is controlled by the Ada Joint Program Office, which also oversees various support activities associated with the language. Originally developed as a programming language, Ada supports a number of modern engineering practices that are also important in system architecture specification. A workshop was held in Alexandria, Virginia, August 11-12, 1986 where representatives of the SDI architectural community joined in pursuing how Ada might best be

[1] In fact, SADMT was originally called the SDI Ada Process Description Language; however, this caused such uproar from certain camps that the appellation has been modified. Admittedly, the new name more accurately describes the purposes of SADMT.

[2] Key words and phrases here are the word "formal" (which precludes the carrying of semantic information in informal comments) and the phrase "Process Description" (which clearly indicates the desire to deal with all systems-level processes--not only computer software programs). The clear intent was to find a way to capture process function and interface requirements at a relatively abstract level, but using Ada constructs as a basis.

used as the basis for the SDI Architecture Process Description Language. As a result, several deficiencies were identified; specifically, Ada facilities in the following four areas were found to be wanting (for direct use as a Process Description Language):

(1) abstract processes,

(2) abstract interprocess communications,

(3) specification of resource bindings from abstract processes to components of the specified execution environment, and

(4) behavioral specification and assertions.

After the workshop, two concurrent tasks were initiated. IBM Federal Systems Division began a translation of their BM/C[3] architecture into IBM Ada PDL. This experiment did not identify any additional deficiencies in Ada as a basis for a Process Description Language. However, the translation of the architecture into a formal notation did identify several aspects of the architecture that were incompletely specified. This result supports the need for a standard formal notation.

Concurrently, IDA began the development of a "strawman" SDI Ada Process Description Language.[3] The next version, SA/PDL version 1.0, was designed by modifying the initial strawman according to suggestions received from the invited reviews and also to follow more closely the general principles outlined below. Based on further public comment, invited reviews, and the initial prototyping effort, the current version, SADMT version 1.50, was defined. Differences between SA/PDL version 1.0 and SADMT version 1.50 are the following:

(1) The interface with the simulation driver for physical events has been specified, and its specification is included here and has become part of the SADMT specification.

(2) Several simple syntactic changes were required to implement the required simulation driver. Also, several new forms and parameters considerably enrich the process-switching (*i.e.*, "wait") primitives.

(3) Resource assignments have become procedural specifications rather than annotated specifications. This is due to the fact that such resource assignments actually affect (and in some cases, effect) the semantics of the function being specified and should not, therefore, be specified in "virtual code" or "annotations."[4] Also, all such assignments are now specified in terms of the SADMT processes and not (directly) in terms of Ada constructs.

(4) A new and different set of annotations has been designed that are (again) in terms of the SADMT process model and not in terms of Ada constructs.

It is assumed that readers of this document are familiar with the Ada language and the notation used for its definition. The same notation is used in this document except that "<>" are used to enclose nonterminal symbols. The reader should consult the *Reference Manual for the Ada Programming Language* [LRM 83] for definitions of nonterminal symbols, such as <expression>, that are not defined in this document.

1.3. General Principles

In this section, the general principles that guided the design of SADMT are enumerated and discussed.

---

[3] Again, this is taken to mean a language or technique for describing abstract SDI architectural processes using the typing and functional constructs of Ada.

[4] Several terms are introduced here without definition so that those readers already familiar with SA/PDL version 1.0 can compare the old version with the new. Other readers should not concern themselves with these points; each term is defined at its proper place in the specification.

Ada Compilable

The most widely available tool for the analysis of designs based on Ada is an Ada compiler. For this reason, SADMT adheres to the grammatical conventions of the Ada language itself. A SADMT description of an architecture is directly compilable by any Ada compiler. In this way, existing compilers may be used to check the consistency of interface specifications. In fact, stubs are included in this document that will allow an Ada compiler to perform some consistency checking on the design even if the simulation driver packages are not available.

The chapter entitled "Behavioral Constraints" describes how SADMT uses the constructs of Ada to capture information about the correct functioning of the system. Ada code used for this purpose is called **virtual code**. The specifications of the constraints themselves (that reference the virtual code) are carried in special structured comments called **annotations**. A tool called ToolA (*i.e.*, a Tool for processing Assertions) is being implemented that accepts as input an annotated Ada program *representing a SADMT description* and that produces as output an Ada program to which code has been added to check the constraints specified in the annotations.

Ada Executable

A SADMT description of an architecture is completely represented by an Ada program and this Ada program, in conjunction with the Simulation Framework, may be executed to simulate the architecture. Thus, SADMT provides a direct interface for simulation and evaluation. As will be seen in the next chapter, SADMT supports an abstract process model and an abstract view of interprocess communication. However, due to the requirement that the description be directly executable, SADMT process descriptions are significantly more verbose than if direct execution were not required. Because of this, a less verbose language has been designed and an accompanying translator, called SAGEN (for SADMT Generator), has been implemented to facilitate the development of SADMT descriptions [Kappel87].

A Description Language

SADMT is designed for describing SDI BM/C^3 architectures. Its primary purpose is to define the interface between the SDI architecture contractors and the evaluation/simulation facilities of the SDIO. SADMT may not be an optimum representation for early architectural development purposes. Rather, different contractors will likely use such methodologies (and require different tools to support these methodologies) as are appropriate for their individual development strategies. However, the resulting descriptions must then be translated into SADMT so that the descriptions may be incorporated into a larger system description. Because of this goal, SADMT was designed (1) to support a general process model similar to those of many automated design tools and (2) to support easy automated generation by design tools. Of course, delivery of any methodology-specific representations is also needed together with any other documentation or representations generated by the tools that were used to develop the architecture.

Some contractors may wish to use an Ada Design Language (ADL) as their primary representation and to base their tools on this representation. Here, SADMT may be found somewhat wanting because its goal is not to facilitate design but rather to capture the *result* of the design effort. As such, it does not include Ada extensions that may be useful in a design (as opposed to description) language. However, the finished product should be formal--free from any informalities that were included during its development; this final description is what SADMT is designed to capture. Therefore, contractors can use the design language (in this case, their ADL) and tools that best suit their methodology.

1.4. Overview

SADMT/SF lets the user describe a system architecture and BM/C^3 architecture for simulation. A SADMT architecture description is composed of several parts. First, the components of the architecture are described as **platforms** moving in space and communicating via **cones**. Platforms are used to model all physical entities such as a ground station, a sensor satellite, a reentry vehicle, or a fragment of debris. Cones are used to model communications between platforms as well as any other "wave-like" items such as a laser beam.

At the next level of decomposition, SADMT/SF represents a platform as a hierarchy of communicating **processes**. In other words, a platform consists of several processes; each of these processes may consist of several other processes, and so on. Finally, at the lowest level, a SADMT platform is a group of leaf processes (processes that are not decomposed) and a network of communication links between these processes. These SADMT processes are used to represent the logical processing associated with a platform as well as models of technology such as sensors and weapons.

In addition to the platforms and processes, SADMT allows the architect to place **constraints** on the communications and performance of the processes. SADMT also allows the designer to specify the mapping of processes to **real resources**. The constraints and real resource assignments are used to check the design and modify the performance to reflect the available hardware.

The basic building blocks of a SADMT description are the processes and communication links. These two components are used together to construct representations of platforms. The platforms are then grouped together to form an initial configuration of the architecture, and a simulation is produced when a configuration of platforms is executed under the control of the SADMT/SF driver.

1.5. SADMT Tool Suite

This section identifies the components of the SADMT system and specifies the capabilities that the use of SADMT will enable as the programs for processing SADMT become available. The intent is to make explicit what these tools are and how they will be used. The SADMT tool suite consists of the following codes:

(1) an Ada compiler,

(2) Ada packages for simulation and evaluation, and

(3) ToolA.

Figure 1-1 shows what capabilities can be obtained and also the tools required. At the top left, an architecture coded according to the SADMT rules is created. The first vertical path in the diagram shows that a SADMT architecture process descriptions and the current packages for the simulation driver can be combined with an Ada compiler to obtain syntactic consistency checking. When the SADMT description for an entire architecture is complete, the architecture may be simulated as shown by the second vertical path in Figure 1-1. Also, the simulation packages provide a more thorough interface check than is possible with just syntactic criteria. The third vertical path shows that a more faithful simulation is possible with the simulation driver packages that support the use of information about the execution environment. The rightmost vertical path shows that the simulation is further enhanced with constraint checking by ToolA. Currently all elements of the system in Figure 1-1 are available with the exception of ToolA. It should be noted, however, that the use of SADMT for system specification and simulation does *not* depend on its early availability of ToolA.

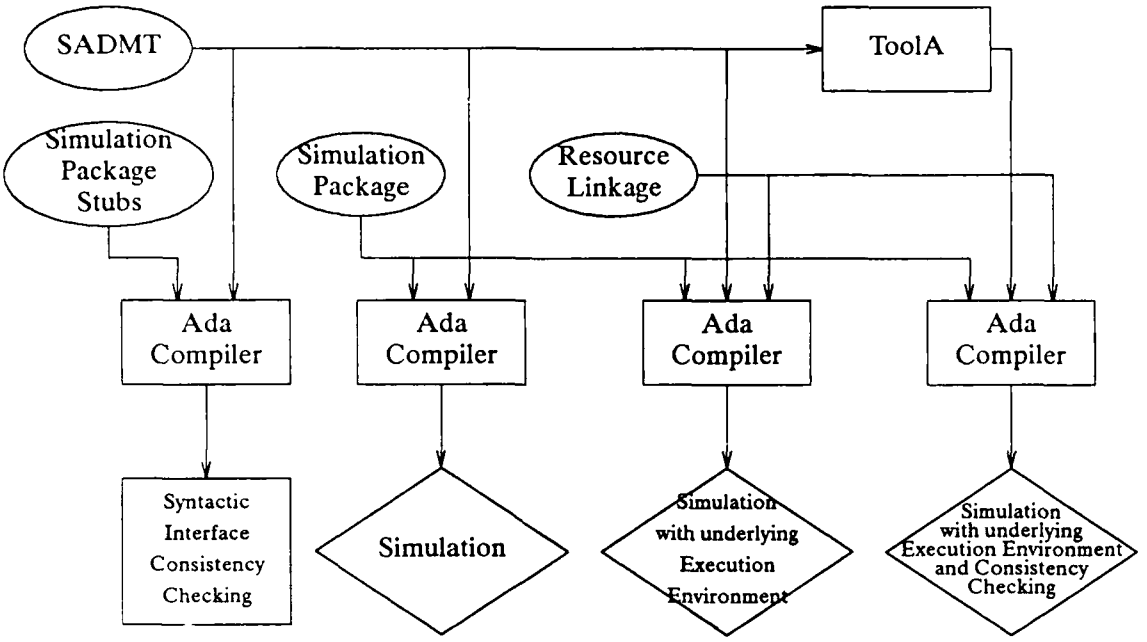


Figure 1-1. Capabilities of the SADMT Tool Suite.

1.6. Usage

SADMT has been developed to describe information processing architectures for the purposes of simulation. It is not intended for use as a design tool (*e.g.*, as a "user friendly" device that engineers use to input designs). Also, it does not specify a standardized way to describe all physical characteristics of a system. Thus, a user who wants to describe system elements in terms of their weight, shape or color will have to extend the language presented in this report. (Actually, non-BM/C^3 processes in the SADMT/SF may represent the effect of these characteristics.) SADMT also allows for the description of information processing components and their allocation (statically and dynamically) to platforms and processing/communications architectures.

SADMT can be used to describe architectures at multiple levels of elaboration. It is not intended to represent concepts prior to their elaboration into defined processes. Since the language allows hierarchic decomposition of processes, as discussed in Chapter 2, these processes can be expressed at whatever level of granularity is appropriate. Similarly, Ada code is the mechanism used to express the behavior of process "bodies." Different levels of code elaboration will allow behavior to be expressed at appropriate levels of fidelity.

It is expected that architecture design contractors will use automated engineering design tools to produce SADMT. Design organizations typically have one or more Native Design Languages (NDLs) that are consistent with that organization's design methodology. The tools used to support the methodology should be capable of automatically producing both SADMT and any standard display format (*e.g.*, $IDEF_0$).

1.7. Organization

This document presents the basic parts of SADMT in four Chapters. Chapter 2 details the basic building blocks, the process model and the interprocess communications model. The basic blocks are put together in Chapter 3 to form platforms. Chapter 3 describes both the structure of a platform and the interface between platforms and the simulated environment. The

methods for constraining the behavior and characteristics of SADMT constructs (processes and ports) is given in Chapter 4; and finally, Chapter 5 describes the method used to specify the resource assignments.

CHAPTER 2

Processes

The foremost purpose of SADMT is to capture architectures in early stages of development for purposes of simulation and evaluation. To provide this capability, SADMT defines an abstract entity called a "process" and a mechanism for specifying a process as a set of communicating subprocesses. The SADMT model of processes requires that processes operate by sending and receiving messages to other processes and that there be no interprocess communications except via this message interface. The message interface between any two processes is strongly typed both with respect to the domain of message values and with respect to the windows, or ports, through which messages are passed. (In other words, each such port has a specific data type associated with it and only data of that type may enter or leave a process through that port.)

When using SADMT, one must use the SADMT process model as the basis for the architectural description and not the (concurrent) model of the underlying representation language–Ada, in this case. SADMT processes are described in Ada according to a very specific template using Ada tasks. There is no requirement, however, that the simulation driver use the low-level Ada tasking primitives for interprocess communications and process switching as long as the SADMT process model is implemented faithfully.[1] In particular, since SADMT processes may only communicate via the SADMT port mechanism, it is required that Ada tasks that represent SADMT process *not* rendezvous or otherwise communicate with another task representing another SADMT process *except via the port mechanism*. Thus, an Ada program representing a SADMT description uses the normal syntax and semantics of Ada augmented by procedure calls to implement the SADMT-specific functions; in this way, an Ada execution system may be used to simulate a SADMT description.

2.1. The SADMT Model of Processes

In SADMT, a system is viewed as a hierarchy of processes. From a hierarchical point of view, there are two kinds of processes: leaf processes and nonleaf processes. Leaf processes are not decomposable in the space of SADMT processes; we assign leaf processes a level number of zero. A nonleaf process is constructed from a set of subprocesses by specifying port connections for each subprocess. The nonleaf process so constructed has level number n+1, where the maximum level number of all of its subprocesses is n. (The level number of a process is only useful in establishing that processes are defined hierarchically; as such, no further mention is made of the level numbers.)

As mentioned above, SADMT processes are defined to have "ports" (*i.e.*, windows) for passing data into and out of a process. All interprocess communication is accomplished via these ports. A port is either an input port or an output port; SADMT makes no provision for bidirectional ports. Further, ports in SADMT are typed; the type of information that can flow into or out of a port is strictly controlled.

Since each nonleaf process is defined as a network of communicating subprocesses, SADMT also defines how these interconnections, or *links*, are specified. There are three types of links: (1) internal, (2) input-inherited, and (3) output-inherited. In all cases, the information type of the ports connected must be the same. An internal link *l*=(p,q) connects output port p of

[1] Actually, the only existing driver is written completely in Ada. A custom driver is certainly a reasonable approach to increase the performance; of course, this sacrifices portability.

subprocess x to input port q of subprocess y where x and y (not necessarily distinct) are immediate children of the process in which *l* is defined. The concept is that data that flows out of output port p flows into input port q. An input-inherited link *l*=(p,q) specifies a correspondence between an input port p on the parent process and an input port q on a child subprocess; for an input-inherited link, the concept is that data directed to input port p of the parent actually flows into the child's input port q. If the link is output-inherited link *l*=(p,q), the correspondence is from an output port p on a child subprocess to an output port q on the parent process; for an output-inherited link, the concept is that data flowing from the child's port p is actually directed to wherever the parent's port q is connected.

An example that shows the various types of links is depicted in Figure 2-1. In Figure 2-1(a), a BM/C^3 process is depicted with a single input port and a single output port. (Note that port names and types are not indicated.) In Figure 2-1(b), the BM/C^3 process has been described as an interconnected set of three subprocesses, (1) threat assessment, (2) a weapon assignment, and (3) preceived view of the world. The links shown in solid lines are internal; these are completely isolated from the external structure that uses the BM/C^3 process. The input port of the BM/C^3 process is shown to be connected (by a dashed line) to the input port of the threat assessment process. Thus, whatever data comes into the BM/C^3 is actually delivered directly to the threat assessment process. Similarly, the output port of the weapon assignment process is linked to the output port of the BM/C^3 process; thus, the output of the weapon assignment process goes into whatever the output of the BM/C^3 process is connected to. For example, if the BM/C^3 process were connected to a weapon launcher process, the output of the weapon assignment process goes into an input port of the weapon launcher.

Port linkages are actually more complicated than indicated by this simple example because of two factors:

(1) any port may participate in several links, that is, an output port can be internally linked to several input ports or linked with inheritance to several output ports of the parent process or any combination of the two. Also, an output port may be unconnected; in this case, data emitted from the port is simply discarded.

(2) data may flow up several levels through output-inherited links, across an internal link, and down several levels through input-inherited links to reach the input ports that are to receive it.

These may be understood more readily by using the terms "data-connected," "fan-in," and "fan-out." First, we say that port p is data-connected to port q through link *l* exactly when there is a sequence of ports $r_0, r_1, \ldots, r_n$, and i, with $0<=i<n$, so that

(1) r_0=p and r_n=q,

(2) $L=(r_i,r_{i+1})$ is an internal link,

(3) $O=(r_j,r_{j+1})$ is an output-inherited link, $0<=j<i$, and

(4) $I=(r_j,r_{j+1})$ is an input-inherited link, $i<j<=n$.

For any output port p, the number of input pairs (q,*L*) where p is data-connected to q through *L* is called the fan-out of p. Similarly, for an input port q, the fan-in of q is the number of pairs (p,*L*) where p is data-connected to q through *L*.

These concepts are available to the user of SADMT in the following way. As one may imagine, there is a primitive that is called *emit* that allows a process to transmit data via one of its output ports, say p. The number of processes that will receive the data is exactly those that are data-connected to p via some internal link; a port that is data-connected to p via more than one internal link will receive multiple copies of the message. Note here that multiple inherited paths to the data-connecting internal link do not contribute to the fan-in or fan-out.

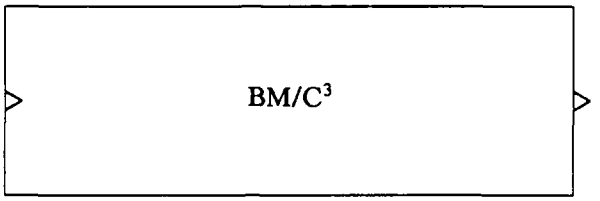


Figure 2-1(a). Depiction of a BM/C^3 Process

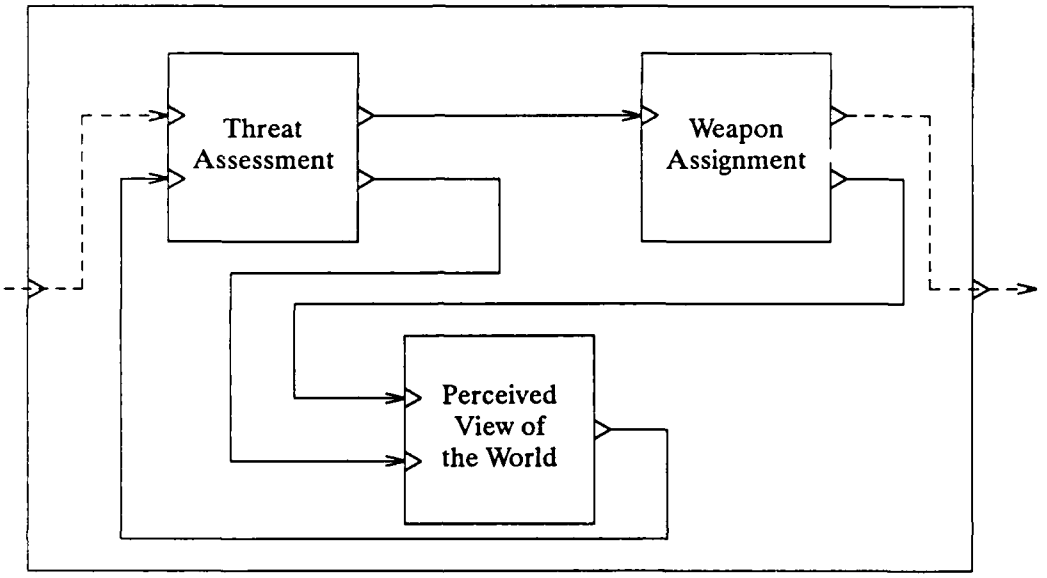


Figure 2-1(b). Exploded View of BM/C^3 Process

There is a special type of port, called "selectable" which differs from regular ports in two ways. First, data emitted to a selectable output port may be directed to a particular set of data-connected input ports. This facility is provided through a special form of the *emit* primitive. Second, multiple paths from a selectable output port to a selectable input port are **not** allowed. This restriction ensures that multiple messages will not arrive at the designated input ports. An example of selectable ports is given in the next section.

As will be discussed subsequently, SADMT provides annotations that allow the specification of port constraints. While deferring the discussion of how the constraints are specified, it is useful here to consider fan-in/fan-out constraints to establish a number of interesting restrictions. For example, one might mandate that the maximum fan-in for any port p be unity. This restriction ensures that for any input port, there is a unique output port that supplies data for that port. (It also ensures that the path by which the data flows is unique.) Conversely, the case of unrestricted fan-in and restricted fan-out is the case best handled by the Ada rendezvous mechanism. SADMT does not mandate any particular policy on fan-in/fan-out constraints.

2.1.1. Representing SADMT Processes in Ada

The following information must be specified for each SADMT process:

(1) the various types of data that flow on internal arcs.

(2) the "semantics" of the process in terms of port activity.

(3) the number, kind, and name of each subprocess.

(4) the number, name, information type, direction, and selectable property of each port.

(5) initialization for each subprocess and port.

(6) the internal and inherited links.

As the representation of each of these notions is discussed, it will be helpful to refer to an example. Figure 2-2, Figure 2-3, Figure 2-4, Figure 2-5, and Figure 2-6 together give the description of a relatively simply system. This system being described is an "army". It is composed of a general an arbiter and 1,000 soldiers.

Before describing the example, a distinction between **message types** and **port types** is in order. A **message type** is any user defined type or predefined type which describes the data that is passed between processes via the ports. A **message type** must be known to both the senders and receivers of messages of that type. A **port type** is the type *T_port* resulting from the instantiation of the *PortDefiner_pkg* package with a **message type** (this will be seen in the following examples). Likewise, input port types and output port types are the types *T_ipptr* or *T_sipptr* and *T_opptr* or *T_sopptr* respectively. Thus, a **port type** defines the point at which objects of type **message type** are transmitted and received.

In the following example, a general has the capability of giving orders (message objects of the **message type** *Order*), and a soldier has the capability of receiving the *Orders* given by the general. Thus, a communications link is established between the general and each soldier and the type of information carried on each link is of type *Order*. Furthermore, any one of the soldiers may be designated as the general's private assistant. Therefore, the general must have the capability of giving an order to his assistant without transmitting the order to any of the other soldiers. For this reason, the communication ports between the general and soldiers are "selectable." Each soldier periodically communicates its status, (dead or alive) to the arbiter. Thus, a communications link is established between the arbiter and each of the soldiers and the type of information carried on each of these links is of type *soldier_status*. Furthermore, the arbiter selects one of the live soldiers to be the general's assistant; therefore, a link between the arbiter and the general allows the arbiter to tell the general which soldier is his assistant. The system may be represented graphically as in Figure 2-2.

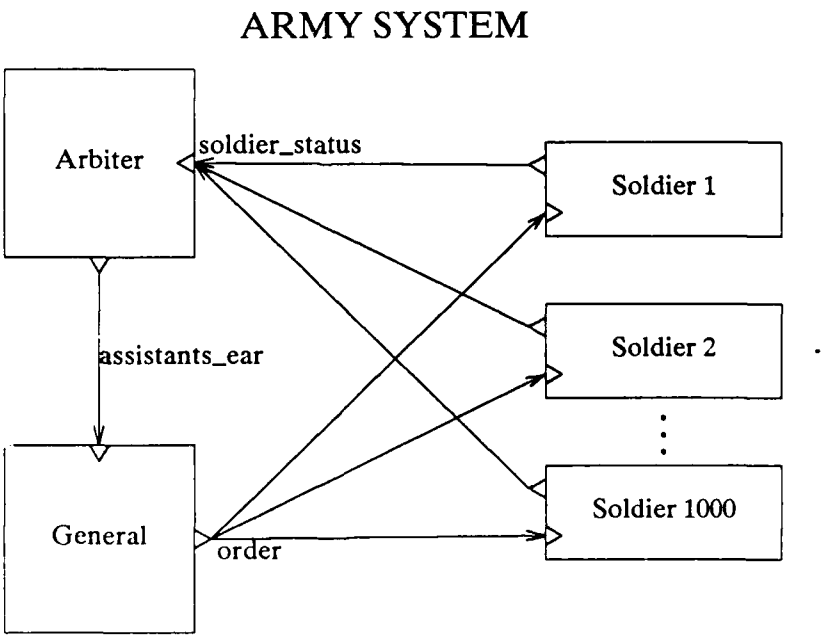


Figure 2-2 Graphical Representation of a Simple Army

Three types of information flows on links in this example: an *Order*, an *Assistants_ear*, and the *Soldier_status*. Concentrating on the *Order* type, it will be necessary here to connect the *give_order* port of the general to the *receive_order* port of each of the soldiers. To do so, we must specify not only a message type (*Order*) but also a port type that can transmit/receive an *Order*: *Order_port*. Due to an Ada restriction, we will also need a pointer to such a port; we use separate access types for input ports and output ports *Order_ipptr* and *Order_sipptr* for input and selectable input ports, and *Order_opptr* and *Order_sopptr* for output and selectable output ports. These types are declared in *Order_pkg*, as shown in Figure 2-3. The *Assistants_ear_pkg*, *Soldier_status_pkg* and *Arbiter_pkg* packages are given in Appendix L.

The generic package *PortDefiner_pkg* (see Appendix A) declares the other five types needed. These are then made visible with subtype declarations. The generic also declares a set of procedures that are personalized for type *Order*.

The *PortDefiner_pkg* package requires four parameters which specify the type of the port, a print procedure to print data of that type, a string to identify the port type, and the amount of memory used internally for port computations. Only the first two parameters must be specified; however, the third parameter is recommended for debugging, and the fourth parameter should be set if the breadth of a communication network of that type is greater than the default value of 200.

```
with PortDefiner_pkg, PDL_pkg;
package Order_pkg is

  type Order is record
    command: integer;
  end record;
  type Order_ptr is access Order;

  Order_debug_class: string(1 .. 5):= "Order";
  procedure put_msg(m: Order; indent: integer:= 35);
  package PD is
      new PortDefiner_pkg(
                Order,
                put_msg,
                "ORDER",
                Order_debug_class);

  subtype Order_port is PD.T_port;
  subtype Order_opptr is PD.T_opptr;
  subtype Order_ipptr is PD.T_ipptr;

end Order_pkg;
package body Order_pkg is
  procedure put_msg(m: Order; indent: integer:= 35) is
    use PDL_pkg.PDL_IO;
    use TXT_IO, INT_IO;
  begin
    for i in 1 .. indent loop
      put(' ');
    end loop;
    put("Order= ");
    put(m.command);
    put_line(". ");
  end put_msg;
end Order_pkg;
```

Figure 2-3. Specification of the Message Type *Order*

2.1.2. The Structure of SADMT Processes in Ada

Consider the actual Ada structure of a SADMT process in Figure 2-4(a), Figure 2-4(b), Figure 2-5, or Figure 2-6. The "semantics" of the process (type) is encapsulated in a task; since the development of these tasks is discussed in the next section, we will not discuss it further here. All of the subprocesses of the process being described, are declared in a record *e.g.*, *Army_subprocesses*. This type should be private since the internal structure of a process should not be visible from outside. A type, called the "process type" (*e.g.*, *Army_block*), is declared that aggregates these various pieces and declares the ports of the process (see either *General_pkg* or *Soldier_pkg*). Last, a procedure must be declared to initialize any instances of the process type.

```
with PDL_pkg, Soldier_pkg, General_pkg, Arbiter_pkg, Order_pkg, Soldier_Status_pkg,
 Assistants_ear_pkg;

package Army_pkg is
  use PDL_pkg, Order_pkg, Soldier_Status_pkg, Assistants_ear_pkg;

  type Army_block;
  type Army_type is access Army_block;

  type Army_subprocesses is private;

  type Army_block is record
    PDL: PDL_ptr:= new_PDL_block(nonleaf);
    SUB: Army_subprocesses;
  end record;

  Army_name: constant PDL_string_ptr:= new string'("Army");
  Army_type_name: constant PDL_string_ptr:= new string'("Army");
  Army_discr_name: PDL_string_ptr:= empty_string;
  Army_characteristic: constant PDL_string_ptr:= new string'("typename=Army");

  procedure initialize(
            z: in out Army_type;
            Parent: PDL_ptr;
            my_name: PDL_string_ptr:= Army_name;
            discr_name: PDL_string_ptr:= Army_discr_name;
            type_name: PDL_string_ptr:= Army_type_name;
            characteristic: PDL_string_ptr:= Army_characteristic);

private
  use Soldier_pkg, General_pkg, Arbiter_pkg;
  type Soldier_vector is array(integer range <>) of Soldier_type;
  type Army_subprocesses is record
    Soldier: Soldier_vector(1 .. 1000);
    General: General_type;
    Arbiter: Arbiter_type;
  end record;
end Army_pkg;
```

Figure 2-4(a). Representation of an *Army* in SADMT

```
package body Army_pkg is
  use Order_pkg.PD.Procedures;
  use Soldier_Status_pkg.PD.Procedures;
  use Assistants_ear_pkg.PD.Procedures;
  use PDL_IO;
  use TXT_IO, INT_IO, TIME_IO, DURATION_IO;
  procedure initialize(
              Z: in out Army_type;
              Parent: PDL_ptr;
              my_name: PDL_string_ptr:= Army_name;
              discr_name: PDL_string_ptr:= Army_discr_name;
              type_name: PDL_string_ptr:= Army_type_name;
              characteristic: PDL_string_ptr:= Army_characteristic) is separate;
end Army_pkg;

separate(Army_pkg)
procedure initialize(
            Z: in out Army_type;
            Parent: PDL_ptr;
            my_name: PDL_string_ptr:= Army_name;
            discr_name: PDL_string_ptr:= Army_discr_name;
            type_name: PDL_string_ptr:= Army_type_name;
            characteristic: PDL_string_ptr:= Army_characteristic) is

begin
  Z:= new Army_block;
  declare
    Soldier: Soldier_vector renames Z.SUB.Soldier;
    General: General_type renames Z.SUB.General;
    Arbiter: Arbiter_type renames Z.SUB.Arbiter;
    MYSELF: PDL_ptr renames Z.PDL;
  begin
    set_process_parent(Z.PDL, Parent, my_name, discr_name, characteristic);
    if init_debug_level > 130 then
      write_process_full(MYSELF, "*init> ", " before start_up");
    end if;
    for i in 1 .. 1000 loop
      initialize(Z.SUB.Soldier(i), Z.PDL, i,
        discr_name => new string'(integer'IMAGE(i)));
    end loop;
    initialize(Z.SUB.General, Z.PDL);
    initialize(Z.SUB.Arbiter, Z.PDL, 1000);

    for i in 1 .. 1000 loop
      internal_link(General.give_order, Soldier(i).receive_order);
      internal_link(Soldier(i).status_out, Arbiter.status_in);
    end loop;
    internal_link(Arbiter.chosen_assistant, General.chosen_assistant);
  -- NONLEAF. Therefore, make_known is placed
  -- in the initialize procedure.
  end;
  make_known(Z.PDL);

  if init_debug_level > 130 then
    write_process_full(Z.PDL, "*init> ", " after start_up");
  end if;
exception
  when others =>
    write_process_full(Z.PDL, "***Some error in ", "***");
end initialize;
```

Figure 2-4(b). Body of an *Army*

The initialization procedure must perform five functions:

(1) "declare" its parent to the SADMT system,[2]

(2) initialize the ports of the process,

(3) initialize the subprocesses,

(4) establish the links, and

(5) if the process is a leaf process, then the task representing the semantics must be started, otherwise make itself known to the SADMT system.

Note that the name *initialize* is heavily overloaded, and it is reasonable to consider here where each of these procedures is declared. The *initialize* procedure for *Army_type*, *General_type*, *Arbiter_type*, and *Soldier_type* are declared in *Army_pkg*, *General_pkg*, *Arbiter_pkg* (see Appendix L), and *Soldier_pkg*, respectively. The *initialize* procedures for *Order* port types are declared in *Order_pkg.PD*. Similarly, the *internal_link* procedure used in *Army_pkg.initialize* is also declared in *Order_pkg.PD*. For this reason, the *Army_pkg*, *General_pkg*, and *Soldier_pkg* packages contain *use Order_pkg.PD.Procedures*.

The argument of *new_PDL_block* tells whether the process being specified is a leaf or not. The significance of this is that the semantics task of any nonleaf process is disabled; instead, the semantic tasks of its descendant leaf processes are executed. The discriminant on an *Order_port* tells whether the port is an input port or an output port, and it tells whether the port is selectable or not selectable. Although it is not indicated explicitly by this example, each datatype for messages has its own version of *internal_link* and *inherited_link*. Thus, a correct Ada program will not result if a port of one type is connected to a port of another type. The parameters of the "link" procedures are also parameterized so that an Ada compiler may automatically check for the correct "in/out-ness" of the links. Unfortunately, the Ada compiler cannot check that two connected ports are in the correct sibling or parent/child relationship; the *inherited_link* and *internal_link* procedures in the simulation driver check for this explicitly.

The results of specifying all of this structural information are:

(1) It provides for the maximum amount of consistency checking by the Ada compiler.

(2) It allows the procedures in the simulation driver to perform even more consistency checking.

(3) It allows the "semantics" of the processes and interprocess communications to be specified in a way that is independent of the actual interconnection of processes. With respect to interprocess communications, a process is only responsible for delivering the data to its ports and receiving data from its input ports; the routines in the simulation driver can assume the responsibility for actually moving the data around. How a process interfaces to others via its ports is the topic of the next section.

2.1.3. Describing the Semantics of SADMT Processes

The task bodies that actually represent the semantics of SADMT processes are simply written in Ada, subject to the following constraints:

(1) The task body must begin with an accept of a *start_up* rendezvous containing a call to make itself known to the simulation's master timing task, as in Figure 2-7.

---

[2] It may seem unusual initially that the child process tells the system who its parent is rather than the reverse. The correct way to think of this is that the parent instructs the child to do this by calling the child's initialization procedure. In this way, there is exactly one action that the parent's initialization must perform for each child. Of course, the port connections are considered separately.

```
with PDL_pkg, Assistants_ear_pkg, Order_pkg;
package General_pkg is
  use PDL_pkg, Assistants_ear_pkg, Order_pkg;

  type General_block;
  type General_type is access General_block;

  task type General_task is
    entry start_up(Z: General_type);
  end General_task;
  type General_task_ptr is access General_task;

  type General_block is record
    PDL: PDL_ptr:= new_PDL_block(leaf);
    SEM: General_task_ptr;
    chosen_assistant: Assistants_ear_ipptr:= new Assistants_ear_port(inport);
    give_order: Order_opptr:= new Order_port(outport);
  end record;

  General_name: constant PDL_string_ptr:= new string'("General");
  General_type_name: constant PDL_string_ptr:= new string'("General");
  General_discr_name: PDL_string_ptr:= empty_string;
  General_characteristic: constant PDL_string_ptr:=
   new string'("typename=General");

  procedure initialize(
              Z: in out General_type;
              Parent: PDL_ptr;
              my_name: PDL_string_ptr:= General_name;
              discr_name: PDL_string_ptr:= General_discr_name;
              type_name: PDL_string_ptr:= General_type_name;
              characteristic: PDL_string_ptr:= General_characteristic);
end General_pkg;
```

Figure 2-5(a). Specification of a *General* in SADMT as a Leaf Process

```
package body General_pkg is
  use Assistants_ear_pkg.PD.Procedures;
  use Order_pkg.PD.Procedures;
  use PDL_IO;
  use TXT_IO, INT_IO, TIME_IO, DURATION_IO;
  task body General_task is separate;
  procedure initialize(
              Z: in out General_type;
              Parent: PDL_ptr;
              my_name: PDL_string_ptr:= General_name;
              discr_name: PDL_string_ptr:= General_discr_name;
              type_name: PDL_string_ptr:= General_type_name;
              characteristic: PDL_string_ptr:= General_characteristic) is separate;
end General_pkg;
separate(General_pkg)
procedure initialize(
            Z: in out General_type;
            Parent: PDL_ptr;
            my_name: PDL_string_ptr:= General_name;
            discr_name: PDL_string_ptr:= General_discr_name;
            type_name: PDL_string_ptr:= General_type_name;
            characteristic: PDL_string_ptr:= General_characteristic) is
begin
  Z:= new General_block;
  declare
    chosen_assistant: Assistants_ear_ipptr renames Z.chosen_assistant;
    give_order: Order_opptr renames Z.give_order;
    MYSELF: PDL_ptr renames Z.PDL;
  begin
    set_process_parent(Z.PDL, Parent, my_name, discr_name, characteristic);
    if init_debug_level > 130 then
      write_process_full(MYSELF, "*init> ", " before start_up");
    end if;
    initialize(Z.chosen_assistant, Z.PDL, "portname=chosen_assistant");
    initialize(Z.give_order, Z.PDL, "portname=give_order");
  end;

  Z.SEM:= new General_task;
  Z.SEM.start_up(Z);

  if init_debug_level > 130 then
    write_process_full(Z.PDL, "*init> ", " after start_up");
  end if;
exception
  when others =>
    write_process_full(Z.PDL, "***Some error in ", "***");
end initialize;
```

Figure 2-5(b). Body of a *General* in SADMT as a Leaf Process

```
with PDL_pkg, Order_pkg, Soldier_Status_pkg;

package Soldier_pkg is
  use PDL_pkg, Order_pkg, Soldier_Status_pkg;

  package Soldier_PARAM_pkg is
    type Soldier_parameterization is record
      Own_index: Integer:= 0;
    end record;
  end Soldier_PARAM_pkg;
  use Soldier_PARAM_pkg;

  type Soldier_block;
  type Soldier_type is access Soldier_block;

  task type Soldier_task is
    entry start_up(z: Soldier_type);
  end Soldier_task;
  type Soldier_task_ptr is access Soldier_task;

  type Soldier_block is record
    PDL: PDL_ptr:= new_PDL_block(leaf);
    SEM: Soldier_task_ptr;
    PRM: Soldier_parameterization;
    receive_order: Order_ipptr:= new Order_port(inport);
    status_out: Soldier_Status_opptr:= new Soldier_Status_port(outport);
  end record;

  Soldier_name: constant PDL_string_ptr:= new string'("Soldier");
  Soldier_type_name: constant PDL_string_ptr:= new string'("Soldier");
  Soldier_discr_name: PDL_string_ptr:= empty_string;
  Soldier_characteristic: constant PDL_string_ptr:=
   new string'("typename=Soldier");

  procedure initialize(
              z: in out Soldier_type;
              Parent: PDL_ptr;
              Own_index_param: Integer:= 0;
              my_name: PDL_string_ptr:= Soldier_name;
              discr_name: PDL_string_ptr:= Soldier_discr_name;
              type_name: PDL_string_ptr:= Soldier_type_name;
              characteristic: PDL_string_ptr:= Soldier_characteristic);
end Soldier_pkg;
```

Figure 2-6(a). Specification of a *Soldier* in SADMT as a Leaf Process

```
package body Soldier_pkg is
  use Order_pkg.PD.Procedures;
  use Soldier_Status_pkg.PD.Procedures;
  use PDL_IO;
  use TXT_IO, INT_IO, TIME_IO, DURATION_IO;
  task body Soldier_task is separate;
  procedure initialize(
              Z: in out Soldier_type;
              Parent: PDL_ptr;
              Own_index_param: Integer:= 0;
              my_name: PDL_string_ptr:= Soldier_name;
              discr_name: PDL_string_ptr:= Soldier_discr_name;
              type_name: PDL_string_ptr:= Soldier_type_name;
              characteristic: PDL_string_ptr:= Soldier_characteristic) is separate;
end Soldier_pkg;
separate(Soldier_pkg)
procedure initialize(
            Z: in out Soldier_type;
            Parent: PDL_ptr;
            Own_index_param  Integer:= 0;
            my_name: PDL_string_ptr:= Soldier_name;
            discr_name: PDL_string_ptr:= Soldier_discr_name;
            type_name: PDL_string_ptr:= Soldier_type_name;
            characteristic: PDL_string_ptr:= Soldier_characteristic) is
begin
  Z:= new Soldier_block;
  declare
    receive_order: Order_ipptr renames Z.receive_order;
    status_out: Soldier_Status_opptr renames Z.status_out;
    Own_index: Integer renames Z.PRM.Own_index;
    MYSELF: PDL_ptr renames Z.PDL;
  begin
    set_process_parent(Z.PDL, Parent, my_name, discr_name, characteristic);
    if init_debug_level > 130 then
      write_process_full(MYSELF, "*init> ", " before start_up");
    end if;
    Z.PRM.Own_index:= Own_index_param;
    initialize(Z.receive_order, Z.PDL, "portname=receive_order",
     "receive_order");
    initialize(Z.status_out, Z.PDL, "portname=status_out", "status_out");
  end;

  Z.SEM:= new Soldier_task;
  Z.SEM.start_up(Z);

  if init_debug_level > 130 then
    write_process_full(Z.PDL, "*init> ", " after start_up");
  end if;
exception
  when others =>
    write_process_full(Z.PDL, "***Some error in ", "***");
end initialize;
```

Figure 2-6(b). Body of a *Soldier* in SADMT as a Leaf Process

```
separate(General_pkg)

task body General_task is
  use timing_ops;
  Z: General_type:= null;

  Assistant: Assistants_ear;
begin
  accept start_up(Z: General_type) do
    General_task.Z:= Z;
    make_known(Z.PDL);
  end start_up;

  declare
    chosen_assistant: Assistants_ear_ipptr renames Z.chosen_assistant;
    give_order: Order_opptr renames Z.give_order;
    MYSELF: PDL_ptr renames Z.PDL;
    package WAITING_pkg is new Wait_pkg(Z.PDL);
    use WAITING_pkg;
  begin
    wait_for_initialization;
    -- the actual semantics of the task. Note that one can access
    -- the ports, PDL, and the parameterization of this task using
    -- the name "Z" or the renaming provide by this declare block.
  end;
end General_task;
```

Figure 2-7. General's Semantics as an Ada Task

(2) The task must not communicate with any other task that represents a SADMT process except under control of the simulation driver. Presently, there are two ways to do this. One is to use the normal port mechanism. The second is to use the "cone" facility. This is discussed in a subsequent section.

(3) The task must be written so that an arbitrary amount of time cannot elapse before the task voluntarily forfeits control to the master timing task. Presently, the only ways to forfeit control are via the *wait* and *wait_for_activity* primitives.

(4) The task must not terminate (*i.e.*, must not complete the execution of the sequence of statements that appears in the body). Thus, the task is either an infinite loop or contains a call to a wait procedure which will never return (*e.g.*, *wait(-1)*).

The first constraint allows the simulation system to find out how many total processes are participating and synchronize them; this information is necessary for setting up its internal data structures for timing. The second constraint ensures that all process communication is performed through ports. The third constraint ensures that process starvation cannot occur; this is fairly standard in simulation systems. The last constraint ensures that processes only forfeit control through the simulation driver. Finally, since the semantics of any nonleaf process is disabled, the task specification and task body are not required (see Figure 2-4(a)).

2.1.4. Port Operations and the PortDefiner_pkg Package

This section examines the details of the *PortDefiner_pkg* package. The topics include, the proper declaration, initialization and use of ports.

Before describing the *PortDefiner_pkg* package, the distinction between **message types** and **port types** should be stressed. A **message type** is any user defined type or predefined type which is passed between processes via the ports. A **port type** is the type *T_port* resulting from the instantiation of the *PortDefiner_pkg* package with a **message type**. Thus, a **port type** defines the

point at which objects of type **message type** are transmitted and received. If the **message types** are not fully defined, one of the type definitions of the *TBD_pkg* package should be used as the **message type**. *TBD_pkg* types should be used only to defer the definition of objects and types whose form or structures is uncertain. The use of *TBD_pkg* types should not prevent the capture of as much of the design as possible in Ada.

```
package TBD_pkg is
  type TBD_TYPE is new INTEGER;
  type TBD_ENUMERATED_TYPE is (A1, A2, A3);
  type TBD_INTEGER_TYPE is new INTEGER;
  type TBD_FLOAT_TYPE is digits 7;
  type TBD_FIXED_TYPE is delta 0.01 range 0.0 .. 1.0;
  type TBD_RECORD_TYPE is record
    null;
  end record;
  type TBD_ARRAY_TYPE is array(INTEGER range <>) of TBD_TYPE;
end TBD_pkg;
```

The *PortDefiner_pkg* package (see Appendix A) is used to define a port type and the operations available for these ports. The first parameter of the generic package is the **message type**, *T*. This type represents the information passed through the port. The second parameter is the print procedure, *print_port_procedure*. This is a user define procedure to print the contents of a message. A generic procedure called *dummy_print_procedure* is provided in the *PDL_pkg* package. This generic procedure may be used if debug information is not needed. An example of its use is given in Appendix L. The third parameter is used to identify the port type for debugging. The name given here is assigned a unique index into the debugging routines for the ports. This gives the user the ability to turn debug information on/off by specifying the *Debug_port_id*. The last parameter is used to control the amount of memory used when performing port computations. In general, this number should represent the maximum breadth of any interconnection network of this port type.

The remainder of this section provides an example illustrating the correct definition, linkage, and use of ports. The complete specification of the *PortDefiner_pkg* is given in Appendix A; therefore, the following will only point to the key issues. We begin by illustrating the definition of a simple port type and the use of that port type. The port type is defined by instantiating the *PortDefiner_pkg* package with a message type. The example of Figure 2-8 defines a message type of *Simple_Msg* and uses this type to instantiate the port type and port procedures.

After a port type is defined, a process may declare ports of that type. Ports are declared within the specification part of a process; therefore, the specification part of a process must "with" the packages which define the the port types of interest to the process. Furthermore, ports are defined as either (1) *inport*, (2) *selectable_inport*, (3) *outport*, or (4) *selectable_outport*. In Figure 2-9, an input port (*message_in*) and an output port (*message_out*) are declared as *inport* and *outport* respectively (see the declaration of *Parent_Process_Block*). The input and output port types, *Simple_Msg_ipptr* and *Simple_Msg_opptr* are defined in the previous example. For this reason, the *Simple_Msg_pkg* package is included in the context clause.

A port must be initialized before it can be connected (linked) to other ports. Initialization and linking are accomplished via the following set of procedures:

(1) *initialize* procedures are used within the initialization procedures of the processes. A port must be initialized before it may be used in any other manner.

(2) *internal_link* procedures are used by a parent process to interconnect the ports of its direct descendants.

```
with PDL_pkg, PortDefiner_pkg;
package Simple_Msg_pkg is
  type route_type is array(1 .. 20) of integer;
  type Simple_Msg is record
    time_created: PDL_pkg.PDL_time_type;
    route: route_type;
    last_slot: integer:= 0;
  end record;
  Simple_msg_Debug_Class: string(1 .. 10):= "Simple_msg";
  procedure put_msg(m: Simple_Msg; indent: integer:= 20);
  package PD is
      new PortDefiner_pkg(
                Simple_Msg,
                put_msg,
                "Simple_msg",
                Simple_msg_Debug_Class);
  subtype Simple_Msg_port is PD.T_port;
  subtype Simple_Msg_ipptr is PD.T_ipptr;
  subtype Simple_Msg_opptr is PD.T_opptr;
end Simple_Msg_pkg;

package body Simple_Msg_pkg is
  procedure put_msg(m: Simple_Msg; indent: integer:= 20) is
    use PDL_pkg.PDL_IO;
    use txt_io, int_io, time_io;
  begin
    for i in 1 .. indent loop
      put(' ');
    end loop;
    put("the message ts,r=");
    put(m.time_created, 1);
    put(":");
    for i in 1 .. m.last_slot loop
      put(m.route(i), 1);
      put(":");
    end loop;
    new_line;
  end put_msg;
end Simple_Msg_pkg;
```

Figure 2-8. A Message Type and Port Definition

```
with PDL_pkg, Simple_Msg_pkg, Simple_Process_pkg;
package Parent_Process_pkg is
  use PDL_pkg, Simple_Msg_pkg;

  type Parent_Process_subprocesses is private;
  type Parent_Process_block;
  type Parent_Process_type is access Parent_Process_block;

  type Parent_Process_block is record
    PDL: PDL_ptr:= new_PDL_block(nonleaf);
    SUB: Parent_Process_subprocesses;
    --DECLARE PORTS
    message_in: Simple_Msg_ipptr:= new Simple_Msg_port(inport);
    message_out: Simple_Msg_opptr:= new Simple_Msg_port(outport);
  end record;

  Parent_process_name: constant PDL_string_ptr:=
   new string'("Parent_Process");
  Parent_process_type_name: constant PDL_string_ptr:=
   new string'("Parent_Process");
  Parent_process_characteristic: constant PDL_string_ptr:=
   new string'("typename=Parent_Process");
  procedure initialize(
              Z: in out Parent_Process_type;
              Parent: PDL_ptr;
              My_name: PDL_string_ptr:= Parent_process_name;
              Discr_name: PDL_string_ptr:= empty_string;
              Process_type_name: PDL_string_ptr:= Parent_process_type_name;
              Characteristics: PDL_string_ptr:= Parent_process_characteristic);
private
  use Simple_Process_pkg;
  type Parent_Process_subprocesses is record
    P1: Simple_Process_type;
    P2: Simple_Process_type;
    P3: Simple_Process_type;
  end record;
end Parent_Process_pkg;
```

Figure 2-9. Specification of *Parent_process*

```
package body Parent_Process_pkg is
  use PDL_IO;
  use TXT_IO, INT_IO, TIME_IO, DURATION_IO;
  use Simple_Msg_pkg.PD.Procedures;

  procedure initialize(
            Z: in out Parent_Process_type;
            Parent: PDL_ptr;
            My_name: PDL_string_ptr:= Parent_process_name;
            Discr_name: PDL_string_ptr:= empty_string;
            Process_type_name: PDL_string_ptr:= Parent_process_type_name;
            Characteristics: PDL_string_ptr:= Parent_process_characteristic) is
    wt1: constant PDL_duration_type:= 20;
    wt2: constant PDL_duration_type:= 30;
    wt3: constant PDL_duration_type:= 50;
    discr_n1: PDL_string_ptr:= new string'("1");
    discr_n2: PDL_string_ptr:= new string'("2");
    discr_n3: PDL_string_ptr:= new string'("3");
  begin
    Z:= new Parent_Process_block;
    set_process_parent(Z.PDL, Parent, My_name, Discr_name);
    initialize(Z.SUB.P1, Z.PDL, Discr_name => discr_n1, waittime => wt1);
    initialize(Z.SUB.P2, Z.PDL, Discr_name => discr_n2, waittime => wt2);
    initialize(Z.SUB.P3, Z.PDL, Discr_name => discr_n3, waittime => wt3);

   -- INITIALIZE AND LINK PORTS
    initialize(Z.message_in, Z.PDL, port_name => "message_in");
    initialize(Z.message_out, Z.PDL, port_name => "message_out");
    inherited_link(Z.message_in, Z.SUB.P2.message_in);
    inherited_link(Z.SUB.P3.message_out, Z.message_out);
    internal_link(Z.SUB.P2.message_out, Z.SUB.P1.message_in);
    internal_link(Z.SUB.P2.message_out, Z.SUB.P3.message_in);
    make_known(Z.PDL);
  exception
    when others =>
      Put_line("**Some error in PARENT_init**");
  end initialize;
end Parent_Process_pkg;
```

Figure 2-10. Body of *Parent_process*

(3) *inherited_link* procedures are used by the parent process to interconnect one of its own ports to one of its direct descendants ports. Both ports must have the same direction.

Like all other port procedures, these procedures are found in the *Procedures* package of the *PortDefiner_pkg* package. Therefore, the body of the process should "use" the *Procedures* package. In Figure 2-10, the body of *ParentProcess_pkg* "uses" the package *Simple_Msg_pkg.PD.Procedures*. The example shows how a port is initialized and linked. In this example, the *message_in* and *message_out* ports of the *Parent_Process* are initialized and linked. The *message_in* port of the *Parent_Process* is linked to the *message_in* port the second *Simple_Process*, and the *message_out* port of the *Parent_Process* is linked to the *message_out* port of the third *Simple_Process*. The remaining two links are internal. These link the *message_out* port of the second *Simple_Process* to the *message_in* ports of the first and third *Simple_Processes*.

After the port is defined, initialized, and linked, it is possible to use the port for communication and synchronization. Since SADMT requires tasks to use ports for interprocess communication, a reasonable set of operations is provided for ports to facilitate developing

appropriate routines. Specifically, the following are provided by the generic *PortDefiner_pkg*:

```
procedure emit(ToPort: T_opptr; Data: T);
procedure emit(ToPort: T_sopptr; Data: T);
procedure emit(
        ToPort: T_sopptr;
        Data: T;
        DestList: PDL_n_PORT_pkg.PortList);
function Port_length(FromPort: T_ipptr) return integer;
function Port_length(FromPort: T_sipptr) return integer;
function Port_empty(FromPort: T_ipptr) return Boolean;
function Port_empty(FromPort: T_sipptr) return Boolean;
function Port_timestamp(
            FromPort: T_ipptr;
            n: integer:= 1)
   return PDL_n_PORT_pkg.PDL_timing.PDL_time_type;
function Port_timestamp(
            FromPort: T_sipptr;
            n: integer:= 1)
   return PDL_n_PORT_pkg.PDL_timing.PDL_time_type;
function Port_data(FromPort: T_ipptr; n: integer:= 1) return T;
function Port_data(FromPort: T_sipptr; n: integer:= 1) return T;
procedure consume(FromPort: T_ipptr; n: integer:= 1);
procedure consume(FromPort: T_sipptr; n: integer:= 1);
```

The following is a brief description of the operations listed above.

(1) *emit* procedures transmit messages across the port linkages. The first two forms of the *emit* procedure transmit the data, *T*, to all ports connected to the outport, *ToPort*. The third form of the emit procedure is used to transmit data to a subset, *DestList* of the selectable ports connected to the outport, *ToPort*.

(2) *Port_length* functions returns the number of messages in the input port's queue.

(3) *Port_empty* functions return TRUE if there are no messages in the input port's queue.

(4) *Port_timestamp* functions return the timestamp of the n^{th} message in the input port's queue.

(5) *Port_data* functions return the information contained within the n^{th} message in the input port's queue.

(6) *consume* procedure removes a message from the input port's queue.

To transmit data through a port, the sending process must call an *emit* procedure. The simulation system will transfer the data passed to the *emit* procedure across the port linkage to the final destinations. The arrival of the data will be timed with respect to the cost (time) associated with the **internal links** it traverses (see Chapter 5). When the data reaches its destined input port, it is queued on the port's message list. The receiving process is now responsible for the data.

The receiving process must make certain that the port's message queue is not empty before calling the *Port_data* function. This can be accomplished through the proper use of the *Port_length*, *Port_empty*, and synchronization functions (*i.e.*, wait procedures presented in Section 2.1.5). After the data is read from the message queue, the receiving process must remove the data from the message queue. This is accomplished through the *consume* procedure. Examples of the proper management of input ports are given in Section 2.1.5.

The remainder of this section is devoted to the port debugging procedures of the *Procedures* package. In the ensuing discussion, a description of each of the procedures is given; the actual specification of these procedures is located in Appendix A.

(1) *put_pptr* procedures print information about a port and the data contained in the port. This procedure will not print the contents of a message passed over the port. The contents of a message can be printed using the *print_port_procedure* of the *PortDefiner_pkg* package.

(2) *Port_Debug* procedures set the DEBUG flag associated with the specified port. Output will not be generated unless the ENABLE flag for the port type is set. The procedure to set and clear the ENABLE flags are located in both this package and the *PDL_pkg* package. If both the DEBUG and ENABLE flags are set, then the system will print a message every time data is sent or received by the port. For an inport, the receipt (*consume*) of a message will be printed, and for an outport, the transmission (*emit*) of a message will be printed along will a list of all ports destined to receive the message.

(3) *Clear_Port_Debug* procedures are used to clear the DEBUG flag associated with a port.

(4) *Enable_Debug_of_Port_type* procedure sets the ENABLE flags associated with all ports created with this instant of the *PortDefiner_pkg*.

(5) *Disable_Debug_of_Port_type* procedure resets the ENABLE flags associated will all ports created with this instant of the *PortDefiner_pkg*.

(6) *Total_Debug_of_Port_type* procedure forces output information to be generated for **all** ports created with this instant of the *PortDefiner_pkg*. **IMPORTANT**: The DEBUG flag associated with the port is ignored by this procedure; therefore, activity on **any** port created with this package is reported despite the setting of the DEBUG and ENABLE flags (This procedure will not alter the value of the DEBUG and ENABLE flags).

(7) *Disable_Total_Debug_of_Port_type* procedure reverses the effect of the *Total_Debug_of_Port_type* procedure.

The port debug routines call the *print_port_procedure* to output the content of a message being sent over the ports.

2.1.5. Synchronization and the PDL_pkg Package

Each SADMT process, which is **not** a technology module nor a platform, should "with" the *PDL_pkg* package (technology modules and platforms "with" the *Cones_n_Platforms* package as shown in Chapter 3). The *PDL_pkg* package is used to access the types, packages, and procedure necessary to the processes of SADMT. The most significant components of this package are the synchronization procedures. SADMT provides a number of primitives for suspension of execution that are intimately related to the process model.

These wait procedures are located in the generic package *Wait_pkg*, shown in Figure 2-11. The generic is parameterized by the *PDL_ptr*; specifically, the current process's *PDL_ptr* (*e.g.*, *Z.PDL*). This implementation approach eliminates the requirement for each process to specify its *PDL_ptr* at each call to a wait procedure.

The first procedure of this package is the *wait_for_initialization* procedure. The *wait_for_initialization* procedure is used to synchronize the processes during initialization. The *wait_for_initialization* procedures call coordinates the initialization of the leaf processes with the start of the platform. Therefore, each leaf processes upon a platform must issue a call to *wait_for_initialization*. The *wait_for_initialization* procedure must be called immediately after the *start_up* rendezvous and the instantiation of the *Wait_pkg* package (see the *General_task* in Figure 2-7). The result of this is that the semantics of the leaf processes are prevented from executing until all of the processes of the platform have been initialized.

```
generic
  Z: PDL_ptr;
package Wait_pkg is
  use PDL_n_PORT_pkg;
  use Resource_Assignment_pkg;
  null_Float_array_1: ArrayOf_Float(1 .. 0);
  null_Float_array: constant ArrayOf_Float(1 .. 0):=
   null_Float_array_1;
  null_PDLstringptr_array_1: ArrayOf_PDLstringptr(1 .. 0);
  null_PDLstringptr_array: constant ArrayOf_PDLstringptr(1 .. 0):=
   null_PDLstringptr_array_1;

  procedure wait_for_initialization(P: PDL_ptr:= Z) renames DSS.wait_for_initialization;
  procedure wait(
          Interval: PDL_duration_type;
          P: PDL_ptr:= Z) renames DSS.wait;
  procedure wait_for_activity(
                  PL: PortList;
                  which_port: out integer;
                  StartTime: PDL_time_type:= PDL_n_PORT_pkg.Current_time;
                  Time_out: PDL_duration_type:= max_PDL_duration;
                  P: PDL_ptr:= Z) renames DSS.wait_for_activity;
  procedure wait_for_activity(
                  PL: PortList;
                  StartTime: PDL_time_type:= PDL_n_PORT_pkg.Current_time;
                  Time_out: PDL_duration_type:= max_PDL_duration;
                  P: PDL_ptr:= Z) renames DSS.wait_for_activity_2;
  procedure wait_for_activity(
                  StartTime: PDL_time_type:= PDL_n_PORT_pkg.Current_time;
                  Time_out: PDL_duration_type:= max_PDL_duration;
                  P: PDL_ptr:= Z) renames DSS.wait_for_activity_3;
  procedure wait(
          Op: PDL_string_ptr;
          NumericParams: ArrayOf_Float:= null_Float_array;
          StringParams: ArrayOf_PDLstringptr:= null_PDLstringptr_array;
          Default_Delay: PDL_duration_type:= 0;
          Time_out: PDL_duration_type:= max_PDL_duration;
          P: PDL_ptr:= Z) renames DSS.processing_delay_wait;
end Wait_pkg;
```

Figure 2-11. Specification of the *Wait_pkg*

All of the remaining procedures stall a process until some activity of interest occurs or until some specified amount of time has elapsed. The first of these, the *wait* procedure, suspends the process for the amount of simulated time denoted by *interval*. A zero-length wait is permitted to simply forfeit control to the simulation driver. For example, a *wait_for_activity* procedure is used when simulating the semantics of a process that is activated by (1) the receipt of data and (2) the fulfillment of some (potentially complicated) condition called a "firing rule." An

interrupt handler is an example of such a process implemented in software. Specifically, a process with three input ports and the firing rule that data is required on all ports to fire can be represented by

```
loop
  if Port_length(z.port1) /= 0 and Port_length(z.port2) /= 0 and
    Port_length(z.port3) /= 0 then
 -- do whatever a firing means
 -- and then .....
     consume(z.port1);
     consume(z.port2);
     consume(z.port3);
  end if;
  wait_for_activity;
end loop;
```

There are several optional parameters to *wait_for_activity*. The *Time_Out* parameter gives the option to wait a certain amount of time for activity but then to regain control if too much time elapses between inputs. Note that if messages are already in the process' queues, the *wait_for_activity* will revive the process after it forfeits control to the simulation driver. The *StartTime* parameter gives the ability not to be awakened for messages until a time in the future. Specifically, messages whose receipt time is less than *StartTime* do not cause activation untile *StartTime*.

The *PL* parameter allows the specification of a subset (actually a list) of the ports that are enabled for causing a wake up; the *which_port* parameter allows the caller to easily determine which port caused the activation. (In case of a tie, the ports are prioritized according to their order in the list.) This may best be explained by an example. Consider a situation where a task has three inports and wants to perform different actions for each port; a fourth action is needed if input is received less often than each 500 time units. The following fragment abstracts the behavior.

```
wait_for_activity((z.port1.PORT, z.port2.PORT, z.port3.PORT), which_port,
 Time_Out => 500);
case which_port is
  when 0 =>
  -- action for timeout
    some_action;
  when 1 =>
  -- action for port1
    some_action;
  when 2 =>
  -- action for port2
    some_action;
  when 3 =>
  -- action for port3
    some_action;
  when others =>
  -- THIS WILL NEVER HAPPEN
    null;
end case;
```

2.1.6. Other Elements of the PDL_pkg Package

The *PDL_pkg* package defines several other useful packages. The first of these are the *PDL_timing*, *timing_ops*, and the *PDL_IO* packages. A complete listing of these packages is located in the appendices.

The *PDL_timing* and *timing_ops* packages define the *PDL_time_type*, the *PDL_duration_type*, several constants, and the operations available upon time types. The *PDL_IO* package makes several IO package visible. These include: *TXT_IO*, for outputting strings and characters; *INT_IO*, for outputting integers; *TIME_IO*, for outputting simulation time values; *DURATION_IO*, for outputting time in terms of durations; *FLT_IO*, for outputting floating point number; *PROCID_IO*, for outputting process identifiers; *PD_IO*, for outputting port directions; and *PNT_IO*, for outputting the type of a process (*e.g.*, leaf).

2.1.7. Process Parameterization

A process parameterization represent a group of local data which may vary from one instance of the process to the next. Furthermore, this data is initialized when the process is initialize (*e.g.*, when the parent process' or platform's initialization procedure call the *initialize* procedure for the process). Figure 2-12(a) and Figure 2-12(b) contain an Ada representation of a *Simple_Process* including the *initialize* procedure. (This package gives the specification for the subprocess used in Figure 2-9 and Figure 2-10. In the *initialize* procedure, the *waittime* is set to the initial value specified in the last parameter. Figure 2-10 of Section 2.1.4 contains the calls to this initialization procedure. In the initialization calls of Figure 2-10, a different value of the parameter *waittime* is passed to each instance of the process *Simple_Processes* (*P1*, *P2*, and *P3*). This will cause each process to behave in a slightly different manner as illustrated in the task body of Figure 2-12(b). Each process will *wait* for a different amount of time depending upon the value of the parameterization variable *waittime*.

2.1.8. Transitions from the October Version

This section enumerates the changes to the Ada representation of SADMT processes since the October 16th draft of Version 1.5. Readers who have not used the October draft should not concern themselves with these points.

(1) The order of the parameters to several of the procedure calls has been modified. Extra packages have been added to help remove the Z.PDLs from the semantics of a process (task bodies). This is accomplished by making the *PDL_ptr* argument the last argument to these procedures. This allows default values to be assigned to these parameters. In particular, a package may be instantiated with the current value of Z.PDL to assign a default value to the *PDL_ptr* parameter; after instantiation, calls to the procedures are no longer required to specify this parameter (see *Wait_pkg* of Appendix B).

(2) The *wait_for_activity* causes a process to forfeit control. The previous release did not always forfeit control.

(3) The parameters to the *PortDefiner_pkg* package have changed. The *PortDefiner_pkg* package now requires a procedure to print the value of an object of type *T*. A generic procedure named *dummy_print_procedure* is provide in the *PDL_pkg* package. This generic procedure should be used when debugging is not desired. Two optional parameters have also been added.

(4) The parameters to *set_process_parent* have changed. This procedure will now accept four string pointers. The first describes the name of the object, and the second discriminates between several object of the same name. The others describe the type and characteristics of the process.

```
with PDL_pkg, Simple_Msg_pkg;
package Simple_Process_pkg is
  use PDL_pkg, Simple_Msg_pkg;

  package Simple_Process_PARAM_pkg is
    type Simple_Process_parameterization is record
      waittime: PDL_duration_type;
    end record;
  end Simple_Process_PARAM_pkg;

  type Simple_Process_block;
  type Simple_Process_type is access Simple_Process_block;

  task type Simple_Process_task is
    entry start_up(Z: Simple_Process_type);
  end Simple_Process_task;

  type Simple_Process_task_ptr is access Simple_Process_task;

  type Simple_Process_block is record
    PDL: PDL_ptr:= new_PDL_block(leaf);
    SEM: Simple_Process_task_ptr;
    PRM: Simple_Process_PARAM_pkg.Simple_Process_parameterization;
     --DECLARE PORTS
    message_in: Simple_Msg_ipptr:= new Simple_Msg_port(inport);
    message_out: Simple_Msg_opptr:= new Simple_Msg_port(outport);
  end record;

  Simple_Process_name: constant PDL_string_ptr:=
   new string'("Simple_Process");
  Simple_Process_type_name: constant PDL_string_ptr:=
   new string'("Simple_Process");
  Simple_Process_characteristic: constant PDL_string_ptr:=
   new string'("typename=Simple_Process");
  procedure initialize(
              Z: in out Simple_Process_type;
              Parent: PDL_ptr;
              My_name: PDL_string_ptr:= Simple_Process_name;
              Discr_name: PDL_string_ptr:= empty_string;
              Process_type_name: PDL_string_ptr:= Simple_Process_type_name;
              Characteristics: PDL_string_ptr:= Simple_Process_characteristic;
              waittime: PDL_duration_type:= 20);
end Simple_Process_pkg;
```

Figure 2-12(a). Specification of *Simple_Process*

```
package body Simple_Process_pkg is
  use PDL_IO;
  use TXT_IO, INT_IO, TIME_IO, DURATION_IO;
  use Simple_Msg_pkg.PD.Procedures;

  task body Simple_Process_task is separate;
  procedure initialize(
              Z: in out Simple_Process_type;
              Parent: PDL_ptr;
              My_name: PDL_string_ptr:= Simple_Process_name;
              Discr_name: PDL_string_ptr:= empty_string;
              Process_type_name: PDL_string_ptr:= Simple_Process_type_name;
              Characteristics: PDL_string_ptr:= Simple_Process_characteristic;
              waittime: PDL_duration_type:= 20) is
  begin
    Z:= new Simple_Process_block;
    set_process_parent(Z.PDL, Parent, My_name, Discr_name, Process_type_name,
     Characteristics);
    Z.PRM.waittime:= waittime;
    initialize(Z.message_in, Z.PDL, port_name => "message_in");
    initialize(Z.message_out, Z.PDL, port_name => "message_in");
    Z.SEM:= new Simple_Process_task;
    Z.SEM.start_up(Z);
  exception
    when others =>
      Put_line("**Some error in SIMPLE_init**");
  end initialize;
end Simple_Process_pkg;
separate(Simple_Process_pkg)
task body Simple_Process_task is
  Z: Simple_Process_type:= null;
  buffer: Simple_msg;
begin
  accept start_up(Z: Simple_Process_type) do
    Simple_Process_task.Z:= Z;
    make_known(Z.PDL);
  end start_up;
  declare
    package WAITING_pkg is new Wait_pkg(Z.PDL);
    use WAITING_pkg;
  begin
    wait_for_initialization;
    loop
      wait_for_activity;
      buffer:= port_data(Z.message_in);
      consume(Z.message_in);
      wait(Z.PRM.waittime);
      buffer.last_slot:= buffer.last_slot + 1;
      buffer.route(buffer.last_slot):= integer(Z.PDL.process_id);
      emit(Z.message_out, buffer);
    end loop;
  end;
exception
  when others =>
    write_process_full(Z.PDL, "AND THEN SOME EXCEPTION in ", "**");
end Simple_Process_task;
```

Figure 2-12(b). Task and Initialization of *Simple_Process*

(5) The method for defining the process Parameterization (*PRM*) record has changed. The Parameterization record is now visible in both platforms and processes. The declarations should be enclosed within a package.

(6) Debug procedures have been added.

(7) The *selectable* port types have been added. This include a special form of the *emit* procedure to handle selectable ports.

(8) The Ada task representing the semantics of a *non-leaf* process is not required.

(9) The *make_known* procedure call that was executed in the semantic task of a non-leaf process must be placed in the *initialize* procedure of the non-leaf process. Since the semantic task is no longer required for *non-leaf* processes, the *make_known* must be moved into the *initialize* procedure.

(10) Unused fields in the process record may be eliminated. The record representing the process is only required to contain a *PDL* field. The other fields, *SEM*, *SUB*, and *PRM*, are only required if they are needed. Therefore, a *non-leaf* process must have a *SUB* field, but does not require a *SEM* field since the semantics of the process are not executed. Likewise, a leaf process must have a *SEM* field, but does not require a *SUB* field.

CHAPTER 3

SADMT/SF, The SADMT Simulation Framework

SADMT is a technique for describing arbitrary systems composed of intercommunicating processes; as such, it is not specific to the problem of describing SDI architectures. However, the SADMT Simulation Framework (SADMT/SF) is specifically designed to provide mechanisms for simulating the processes found in both a Strategic Defense System (SDS) architecture and a threat architecture. These logical processes are (of course) specified using SADMT as described above. The three major thrusts of SADMT/SF are

(1) to implement the semantics of the SADMT process model and the associated underlying system of (simulated) time and space,

(2) to structure a simulation so as to effect a clean separation of the architectural representation from the simulated environment, and

(3) to structure the simulation so as to cleanly separate the processes that deal directly with the environment from the BM/C^3 processes that do not.

The model used by the SADMT/SF to represent the activities in the physical environment for SDI simulation was suggested by Dr. Richard Lipton [Lipton 87]. This model provides two types of entities called platforms and cones. Platforms are used to represent all physical entities including the sensors platforms, weapons, and carrier vehicles of the SDS as well as the weapons and debris of the threat. A platform specification must contain enough information to allow the simulation driver to automatically move the platform around in space as simulated time progresses, this information is called the **equation of motion** of the platform. Cones are the mechanism by which platforms become aware of other platforms in the system. The basic mechanism of a cone is that a platform may "emit a cone" by describing the geometry of the cone and its associated data; this causes the characteristics of the emitting platform and the associated data to become known to all other platforms that fall within the cone. This facility is used, for example, for modeling communication, radar, and laser beam weapons. The SADMT/SF simulates the movement of all platforms and the emission of cones. It determines when and if two platforms collide and if any platform falls within an emitted cone. Possible effects of a collision with another platform include the destruction of the platform, damage to the platform, or a change in the platform's equation of motion. Possible effects of a "beaming"[1] include the return of a radar signal, the receipt of a communication, destruction or damage to the platform, or nothing at all.

Associated with each platform are some number (possibly zero) SADMT processes that are used to represent the BM/C^3 processes of an SDS sensor platform, weapons platform, carrier vehicle, or threat vehicle. These are called the BM/C^3 processes of that platform.

Also associated with each platform are a set of SADMT processes that interface directly with the environment called **technology modules**. They (TMs) are used to represent entities such as sensors, communications between platforms, weapons, and platform accelerators. The ability to reuse technology modules across different architectures is a fundamental aspect of the SADMT/SF.

[1] Beaming is the term used throughout this document to indicate the action of receiving a cone. In other words, a platform receives a beaming when it is located in the path of a cone.

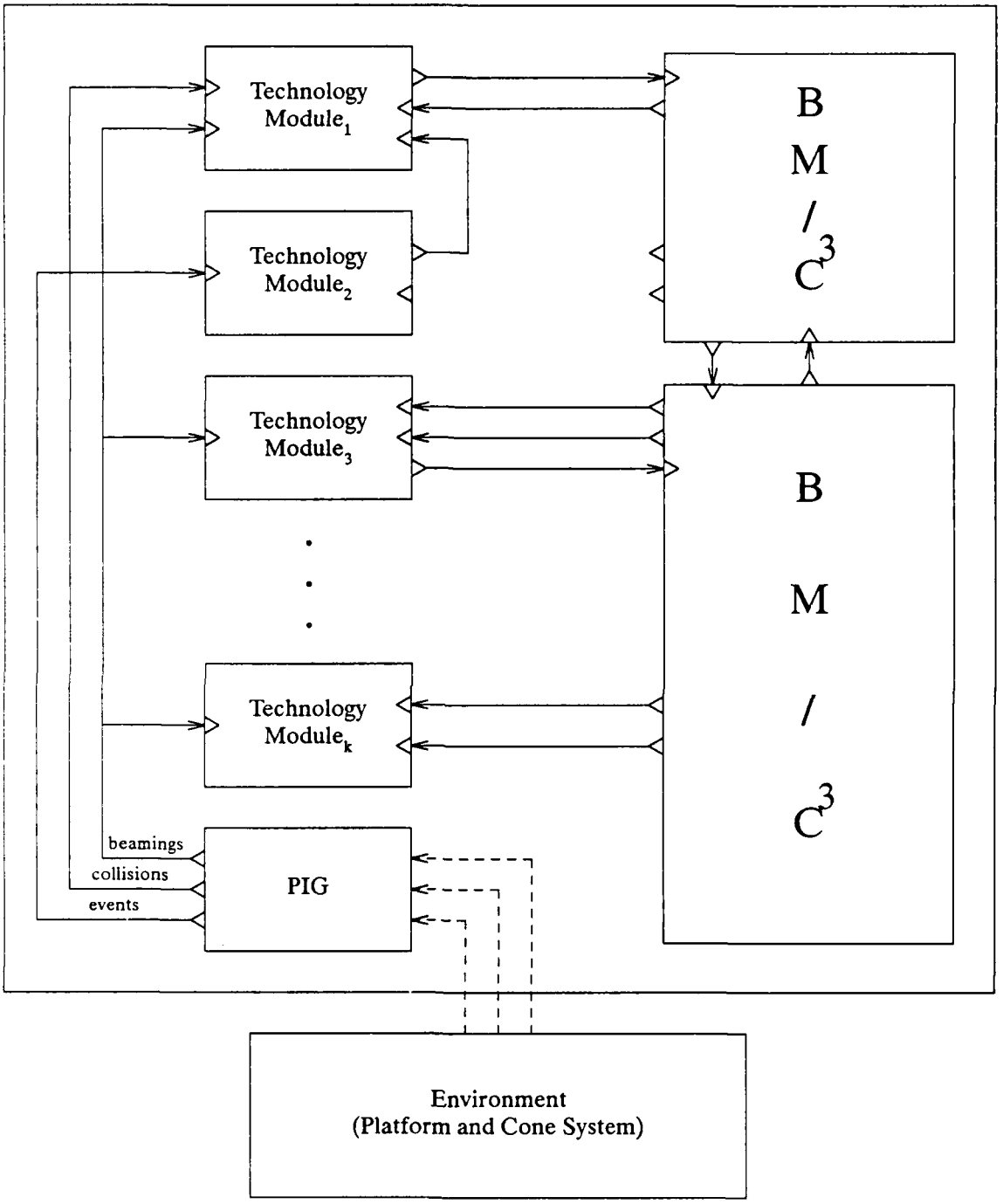


Figure 3-1 High-level View of SADMT/SF

The interfaces of the system are shown in Figure 3-1. The interaction between the BM/C^3 processes of a platform and the simulated physical environment is limited to port interfaces with TMs. These TMs are also In the future it is expected that the specifics of the interface to each TM will be given in an SDI Technology Handbook that documents all SDIO approved TMs. The BM/C^3 processes must adhere to that interface in the same way that a BM/C^3 process of the

SDS will be required to adhere to the physical interfaces of an actual sensor or weapon. The SADMT/SF does not provide these validated TMs or any TMs. It is simply a framework in which they can be used. Thus, early users of the SADMT/SF must provide their own TMs or modify the ones provided as examples with the simulation driver.

The TMs are allowed to interface directly with the physical environment through procedure calls. These procedure calls **cannot** be used by the BM/C^3 process of the platform. (This is enforced by the Ada scoping rules.) In addition, the TMs interface to a *PIG* (Platform InterfacinG) through ports. A *PIG* must be included in each platform and it provides ports through which the SADMT/SF sends messages to TMs indicating that (1) the platform has collided, (2) the platform has been beamed (*i.e.*, it was found to lie within an emitted cone), or (3) the platform's expected lifetime or equation of motion has expired. The dashed lines in Figure 3-1 indicate that the interface from the environment to the *PIG* is not a typical SADMT port interface but a SADMT/SF system interface. Users of SADMT/SF cannot use this interface.

The approved TMs determine and affect the outcome of all collisions. Only a TM associated with a particular platform may directly affect that platform. Thus, a platform cannot directly modify or destroy any other platform. In this way, the simulation of the environment and all assumptions inherent in the simulation are separated from the definition of an architecture. The assumptions, therefore, may be held constant over the evaluation of many architectures, supporting a fair evaluation and comparison of architectural concepts.

Input to the generation of a simulation will include the definition of all platforms in the SDI Architecture and the threat (represented as platforms) as well as the specification of the initial state. Additional platforms and cones will be created by the TMs and platforms may be destroyed as the simulation progresses. An compilable SDS example can be found in the companion document, *A Simple Example of an SADMT Architecture Specification*.

3.1. Platforms

A platform is an aggregated structure consisting of the following parts:

(1) the type of the platform, such as platform, missile, or fired weapon,

(2) the physical characteristics of the platform such as mass, position in the environment, equation of motion, and lifetime,

(3) the SADMT representation of the logical processes of this platform representing (1) the BM/C^3, (2) the TMs, and (3) the *PIG* associated with this platform, and

(4) the SADMT hardware components associated with the platform.

Platforms can be created at the start of the simulation as the initial components of the architecture, or by a TM when a weapon or missile is fired, or when a platform breaks up. A TM can modify the type or physical characteristics of the platform with which it is associated. A TM can also destroy the platform with which it is associated. Importantly, no process on one platform can instruct the driver to modify the characteristics (including destruction) of any other platform. Since such instructions always emanate from the TMs of a particular platform, the problem of determining how a particular platform's characteristics are modified is greatly simplified. The specification and body of a trivial platform are provided in Figure 3-2 and Figure 3-3, respectively.

3.1.1. Representing SADMT Platforms in Ada

There are a few differences between Ada representation of a platform and a non-platform process. First, since the platform interfaces directly with the environment, the package *Cones_n_Platforms* is referenced rather than *PDL_pkg*. Accordingly, this package is "used" before the normal *use* clause sequence. Second, platforms "types" have a string designator associated with them. This string is used to create platforms of this type. (If the Ada type was

```
with Cones_n_Platforms, Simple_Msg_pkg;
with Parent_Process_pkg, RW_Process_pkg, Gyro_pkg;
with UNCHECKED_CONVERSION;
package TopLevel_Platform_pkg is
  use Cones_n_Platforms;
  use PDL_pkg, Simple_Msg_pkg;
  TopLevel_Platform_designator: constant platform_designator_type:=
   new string'("TOPLEVEL");
  TopLevel_name: constant PDL_string_ptr:=
   new string'("TopLevel_Platform");

  type TopLevel_Platform_subprocesses is private;

  package TopLevel_Platform_PARAM_pkg is
    type TopLevel_Platform_parameterization is record
      null;
    end record;
    type TopLevel_Platform_parameterization_ptr is
     access TopLevel_Platform_parameterization;
  end TopLevel_Platform_PARAM_pkg;
  use TopLevel_Platform_PARAM_pkg;
  package TopLevel_Platform_CP_pkg is
   new interface_procs.PlatformDefiner_pkg(
                        T => TopLevel_Platform_parameterization,
                        T_ptr => TopLevel_Platform_parameterization_ptr);
private
  use Parent_Process_pkg, RW_Process_pkg, PIG_pkg, Gyro_pkg;
  use interface_procs;
  type TopLevel_Platform_subprocesses is record
    parent1, parent2, parent3: parent_process_type;
    rw: rw_process_type;
    g: gyro_type;
    PIG: PIG_type;
  end record;
end TopLevel_Platform_pkg;
```

Figure 3-2. Package Specification for a Simple Platform

used, an internal change in a KKV platform, say, would require the weapons platform that creates (fires) the KKVs to be recompiled. This can be contained somewhat by controlling the platform designators and parameterization types.) Third, since platforms may be created dynamically by calls to the simulation driver, the representation of a platform must contain code to provide creation and initialization. Initialization is essentially identical to that of normal processes except for parameterization which is needed for dynamic platform creation and separate compilation. The creation and parameterization structures are discussed below.

3.1.2. Platform Creation and Parameterization

The parameterization of a platform is similar to that of a process. Basically the parameterization represents the data which may vary from one instance of a platform to the next. A simple example is when several identical sensors devices use different keys to encrypt their communications; the parameterization can be used to give each of the sensors a different key when it is created.

The primary difference between a process parameterization and a platform parameterization is the method for assigning a value to the parameterization. A process' parameters are passed into its initialization procedure by its parent. A platform, on the other hand, receives its parameters through a pointer passed into the *create_platform_procedure*. This pointer value

```
package body TopLevel_Platform_pkg is
  use Simple_Msg_pkg.PD.Procedures;

  type TopLevel_Platform_block;
  type TopLevel_Platform_type is access TopLevel_Platform_block;

  platform_designator: platform_designator_id_type;
  function ptr_to_int is
     new UNCHECKED_CONVERSION(platform_designator_type,integer);

  type TopLevel_Platform_block is record
    PDL: PDL_ptr:= new_PDL_block(platform);
    SUB: TopLevel_Platform_subprocesses;
    PRM: TopLevel_Platform_parameterization;
  end record;

  procedure initialize(
            Plat_interface: out PIG_type;
            param: TopLevel_Platform_parameterization_ptr;
            My_name: PDL_string_ptr:= TopLevel_name;
            Discr_name: PDL_string_ptr:= empty_string;
            Characteristic: PDL_string_ptr:= empty_string);

  package Creator is
     new PlatformCreator_pkg(
               TopLevel_Platform_parameterization,
               TopLevel_Platform_parameterization_ptr,
               lookup_platform_designator(TopLevel_Platform_designator),
               initialize);

  procedure initialize(
            Plat_interface: out PIG_type;
            param: TopLevel_Platform_parameterization_ptr;
            My_name: PDL_string_ptr:= TopLevel_name;
            Discr_name: PDL_string_ptr:= empty_string;
            Characteristic: PDL_string_ptr:= empty_string) is
    Z: TopLevel_Platform_type;
  begin
    Z:= new TopLevel_Platform_Block;
    set_process_parent(Z.PDL, null, My_name, Discr_name, characteristic);
    if param /= null then
      Z.PRM:= param.all;
    end if;
    initialize(Z.SUB.parent1, Z.PDL);
    initialize(Z.SUB.parent2, Z.PDL);
    initialize(Z.SUB.parent3, Z.PDL);
    initialize(Z.SUB.rw, Z.PDL);
    initialize(Z.SUB.g, Z.PDL);
    initialize(Z.SUB.PIG, Z.PDL);
    internal_link(Z.SUB.parent1.message_out, Z.SUB.parent2.message_in);
    internal_link(Z.SUB.parent1.message_out, Z.SUB.parent3.message_in);
    internal_link(Z.SUB.parent2.message_out, Z.SUB.rw.message_in);
    internal_link(Z.SUB.parent3.message_out, Z.SUB.rw.message_in);
    internal_link(Z.SUB.rw.message_out, Z.SUB.parent1.message_in);
    Plat_interface:= Z.SUB.PIG;
  end initialize;
end TopLevel_Platform_pkg;
```

Figure 3-3. Package Body for a Simple Platform

enters the platform's *creation* function at the time the platform comes into existence.

Platform creation is accomplished by calls to the procedure *create_platform*. This procedure is located in the generic package *PlatformDefiner_pkg* (shown in Figure 3-4).

This generic package is parameterized on the platforms' parameter type and a pointer to that type. The *create_platform* procedure expects one or more arguments. The following is a brief description of these arguments.

```
--PLATFORM DEFINER PACKAGE
generic
  type T is private;
  type T_ptr is access T;
package PlatformDefiner_pkg is
  procedure create_platform(
                platform_designator: platform_designator_type;
                name: PDL_string_ptr:= empty_PDL_string;
                discr: PDL_string_ptr:= empty_PDL_string;
                param: T_ptr:= null;
                mass: float:= 0.0;
                cross_sect_rad: float:= 1.0;
                initial_position: point_type:= origin;
                eqn_motion: eqn_motion_type:= stay_at_000;
                expected_lifetime: PDL_duration_type:= max_PDL_duration;
                birth: PDL_time_type:= Current_PDL_time);
end PlatformDefiner_pkg;
```

Figure 3-4. *PlatformDefiner_pkg* Package

(1) *platform_designator* is a pointer to a string which uniquely identifies this *type* of platform. This string pointer is used to create platforms of this type.

(2) *name* and *discr* are pointers to strings that are used together to identify a platform. All output from the SADMT/SF about the platform will uses these two pointers are used to compose a string (name) which identifies the platform. These pointers may also used by the resource assignment module described in Chapter 5.

(3) *param* is a pointer to the parameterization for this instance of the platform.

(4) *mass* is the mass of the platform. This is not used by the simulation driver; however, it is very useful to the technology modules.

(5) *cross_sect_rad* is the radius of the sphere which represent the platform. This value is used to determine if two platform collide and if a platform receives a *cone*.

(6) *initial_position* is the point in 3-space where the platform will be created.

(7) *eqn_motion* is a pointer to an equation of motion record. This describes the motion of the platform in space (see Section 3.4).

(8) *expected_lifetime* is the duration used to control the amount of time in which the platform will exist (see Section 3.4).

(9) *birth* is the time when the platform will be created.

3.1.3. Equation of Motion and Lifetime

A platform's equation of motion is represented as a linked list of locations (*positions*) in space and times (*delta_ts*) required to travel from one location to the next (see Figure 3-5). It is assumed that a platform moves at constant velocity between two points. This provides a flexible method for specifying high, low, and variable resolution paths in space.

Figure 3-6 pictorially represents the following equation of motion: a platform travels from point A to B, C, and then D, where the time required to travel between A and B is 10 seconds, B and C is 20 seconds, and C and D is 10 seconds. To describe the motion of such a platform, a list of three points is needed. A sample program is given in Figure 3-7. A more realistic trajectory is given the the companion document, *A Simple Example of an SADMT Architecture Specification*.

```
type eqn_motion_rec;
type eqn_motion_type is access eqn_motion_rec;
type eqn_motion_rec is record
  position: point_type;
  delta_t: PDL_duration_type;
  back_ptr_flag: boolean:= false;
  next_rec: eqn_motion_type:= null;
end record;

package eqn_motion_pkg is
  function new_eqn_motion_rec return eqn_motion_type;
  --this function provides a new record for building
  --representations of equations of motion.
  procedure free_eqn_motion_rec(e: in out eqn_motion_type);
  --this procedure frees a SINGLE eqn_motion_rec, placing it
  --in the global store. If this record points to more used
  --space, that space will NOT be freed, but will be lost.
  procedure free_eqn_motion(e: in out eqn_motion_type);
  --this procedure frees an entire linked list of eqn_motion_rec's.
  --if the list is circularly linked, it transforms it into a
  --flat list in order to free it all at once.
end eqn_motion_pkg;
```

Figure 3-5. Equation of Motion: Types and Functions

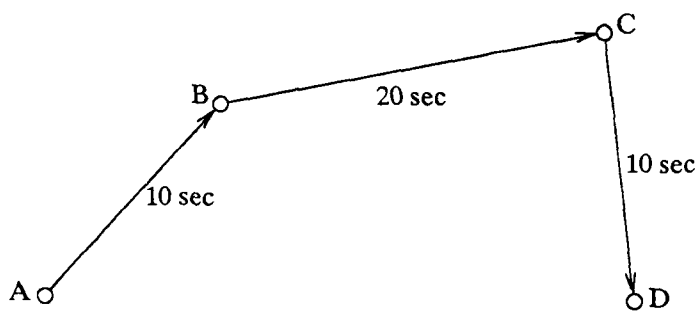


Figure 3-6. Example of an Equation of Motion

```
with System_Scheduler;
with EOM_platform_pkg, Cones_n_Platforms, Vector_pkg;
with VERDIX, Debug_flags;
use EOM_Platform_pkg;
procedure test_EOM is
  use System_Scheduler;
  use Cones_n_Platforms, Vector_pkg;
  use PDL_pkg;
  use PDL_IO;
  use txt_io;
  package IP renames Cones_n_Platforms.interface_procs;
  use eqn_motion_pkg;
  save_eqn, eqn: eqn_motion_type;
  A: point_type:= (0.0, 0.0, 1.0);
  B: point_type:= (1.0, 1.0, 1.0);
  C: point_type:= (3.0, 1.4, 1.0);
  D: point_type:= (3.1, 0.0, 1.0);

  package Create_M_Platform renames EOM_platform_CP_pkg;

begin
  put_line("hello");
  eqn:= new_eqn_motion_rec;
  save_eqn:= eqn;
  eqn.position:= D;
  eqn.delta_t:= 10;

  eqn:= new_eqn_motion_rec;
  eqn.position:= C;
  eqn.delta_t:= 20;
  eqn.next_rec:= save_eqn;
  save_eqn:= eqn;

  eqn:= new_eqn_motion_rec;
  eqn.position:= B;
  eqn.delta_t:= 10;
  eqn.next_rec:= save_eqn;

  Create_M_Platform.create_platform(EOM_Platform_designator,
   initial_position => (0.0, 0.0, 0.0), eqn_motion => eqn, Birth => 0,
   expected_lifetime => 225);
  put_line("simulation begins.....");
  start_simulation(500);
end test_EOM;
```

Figure 3-7. Program to Create a Platform with an Equation of Motion

The function *new_eqn_motion_rec* creates the records needed in an equation of motion. The programmer is responsible for chaining the records of successive calls together to form a linked list. Therefore, after the first call we set the position of point B and the time required to reach point B from point A. Note that the program does not allocate a record for the point A since A will be set to the starting position for the platform by the call to *create_platform*. After all of the records are created and linked, a platform is created at point A with the equation of motion described above. This new platform will travel (**in a straight line**) from location A to location B in 10 seconds, and so on.

When the platform reaches point D (forty seconds after starting) the simulation driver will emit an *Event_Msg* message on the *PIG's events* port. This message will inform the platform that it has reached the end of its equation of motion. At this point, the platform must provide a new equation of motion or the simulation driver will destroy the platform without further notice.

A new equation of motion can be provided by a technology module. Such a technology module would build a list and provide that list to the simulation driver via a call to the procedure *change_my_eqn_of_motion* (see Appendix D). The technology module is also responsible for releasing the space used by the old equation of motion. The simulation driver provides routines to assist in creating and managing equation of motion list; but alas, the technology module is ultimately responsible. Lastly, if the path of a platform is cyclic or stationary, then a circular list can be constructed by setting the *back_ptr_flag* of the *eqn_motion_rec*.

Each platform has a birth time and an expected death time. These values are set by the call to *create_platform*. The platform is created when the simulation time reaches the birth time, and the platform will receive a message from the *PIG* (on the *events* port) when the simulation clock reaches the platforms death time. If the platform does not extend its expected lifetime, the simulation driver will remove it from the system in the next clock cycle.

A technology module may extend the expected lifetime of a platform by calling the procedure *extend_my_lifetime*. A call to this procedure will length the expected lifetime of a platform by the amount of simulation time specified.

The simulation driver also provides a means for determining the expected life time of a platform. A technology module may call the function *get_my_deathtime* to discover the expected life time of a platform.

3.2. Cones

A cone can be created by a TM for communication purposes, for radar sensing, or upon the detonation of a weapon. The Ada type used to represent cones is given below.

```
type Cone_Msg is record
  designator: cone_designator_type;
  initiator_id: Cone_RetAddr_type;
  cone_characteristics: cone_type;
  data: PDL_magic_ptr;
end record;
```

The type *Cone_Msg* is the message type carried over the *PIG*'s *beamings* port to the technology modules of the platform. The simulation driver represents a cone's type (message type) in four parts. The parts are the *designator*, the *initiator_id*, the *cone_characteristics*, and the *data*. The *data* field is where the information specific to this cone type is stored. The *designator* is a string pointer to a name which identifies the type of the cone. This is important to SADMT since the *PIG* only has one port to convey the beaming to the platform. As a consequence, the technology modules connected to the *beamings* port of the *PIG* must use the *designator* to convert the *data* (*PDL_magic_ptr*) to the appropriate Ada type.

The *initiator* indicates the source of the cone (*i.e.*, the platform that created the cone which caused the beaming). By knowing the platform which originated the cone, the technology modules receiving the cone can determine useful information about the location and distance of the sending platform. The *cone_characteristic* describes the direction and spread of the cone. The type *cone_type* is described below.

The direction, and distribution of a cone is determined by the sender. The Ada type used to represent the characteristics of a cone is:

```
type cone_type is record
   source_point: point_type;
   indicator_point: point_type;
   half_angle: float;
   blackout_radius: float:= 0.0;
end record;
```

Figure 3-8 illustrates the components of the *cone_type*. The *source_point* in combination with the *indicator_point* determines the axis (direction) of the cone. The *half_angle* determines the spread of the cone, and the *blackout_radius* is the **minimum** distance a platform must be from the source point to receive the cone. The default for *blackout_radius* is zero but it can be changed to model more complex situations such as shadowing.

Shadowing with two cones is shown in Figure 3-9. In this illustration, one cone is used to represent the "real" cone and the other is used to cancel the effect of the first. The original (real) cone is created from the source on the left side of the figure to the right side of the figure. This cone strikes a platform which obstructs the cone's transmission. To simulate the effect on the transmission, the obstructing platform emits an "anticone" (one which will cancel the effect of the previous cone) using the same *source_point* as the original cone and its own location as the *indicator_point*; furthermore, the obstructing platform will set the *blackout_radius* to the distance between the *source_point* and itself. The *blackout_radius* prevents platforms between the source and the obstruction form receiving the anticone. Platforms that receive the original cone and one or more anticones will ignore the original cone, and platforms that receive only the original will process the cone accordingly.

When a platform creates a cone it must specify all of the characteristics of the cone. The function *platform_position* and the package *Vector_pkg* are useful when specifying the characteristics of a cone.

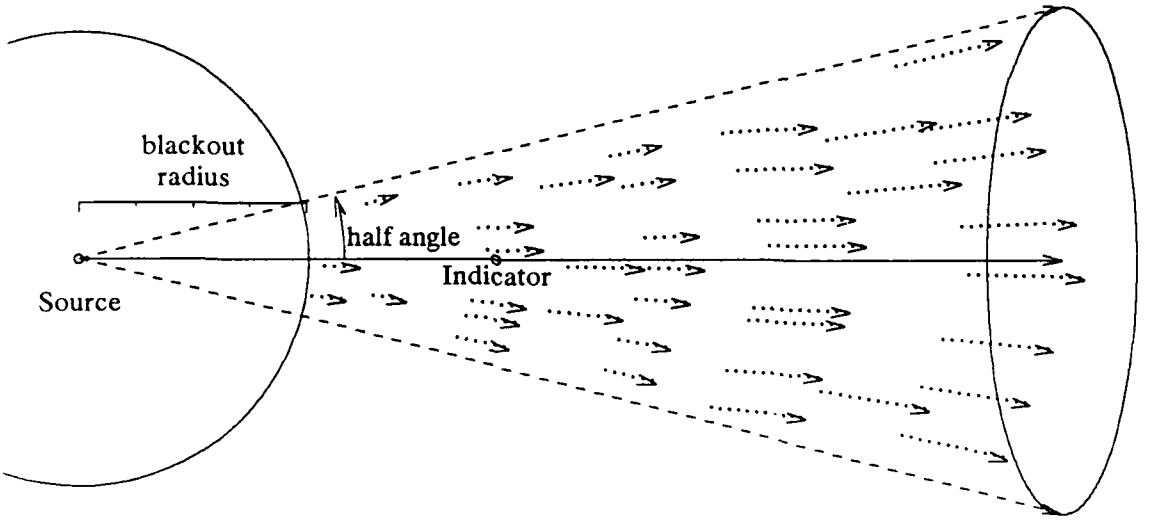


Figure 3-8. Characterization of a Cone

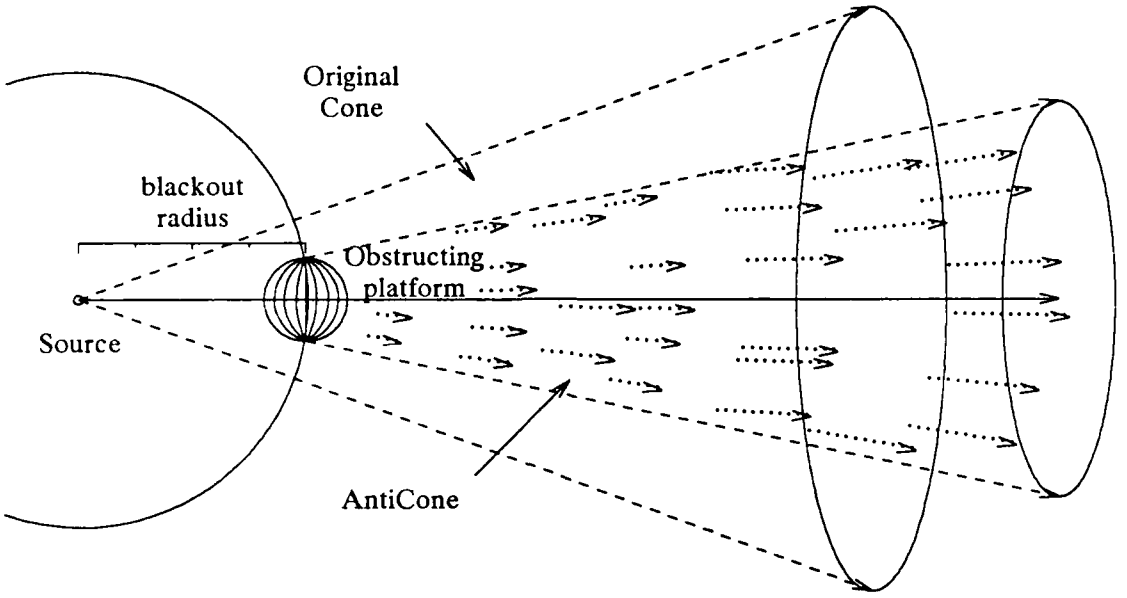


Figure 3-9. Shadowing with Two Cones

The specification of the *ConeDefiner_pkg* package is given below:

```
---CONE DEFINER PACKAGE
generic
   type T is private;
   type t_ptr is access T;
package ConeDefiner_pkg is
   procedure create_cone(
                cone_designator: cone_designator_type;
                cone: cone_type:= cone_everything;
                data: T_ptr:= null;
                Z: PDL_ptr);
   procedure create_cone3(
                 cone_designator: cone_designator_type;
                 RetAddr: Cone_RetAddr_type;
                 data: T_ptr:= null;
                 Z: PDL_ptr);
   procedure create_cone(
                cone_designator: cone_designator_type;
                RetAddr: Cone_RetAddr_type;
                data: T_ptr:= null;
                Z: PDL_ptr) renames create_cone3;
end ConeDefiner_pkg;
```

The first procedure creates a cone according to the characteristics specified in the parameter *cone*. The second procedure creates a special type of cone. This type is used to provide direct communication between two platforms. This can be used to reduce the simulation overhead in several situations (*i.e.*, radar returns).

3.3. Technology Modules

Technology modules are the only processes that may interface directly with the environment. Thus, a platform consisting of technology modules and BM/C^3 processes contains a sharp separation between the architectural representation of BM/C^3 and the simulated environment.

The technology modules of a platform are represented as SADMT processes. Their responsibility is to manage the interaction between the BM/C^3 processes of the platform and the simulated physical environment. BM/C^3 processes and technology modules interface through SADMT ports; technology modules interface to the simulation system via a combination of ports and procedure calls.

Technology modules receive input from the SADMT/SF through three output ports connected to the *PIG* (Platform InterfacinG) process. Every platform is required to have a *PIG* process. As a simulation progresses, events of potential interest to a platform are indicated by the arrival of messages on the *PIG*'s output ports. The message emitted on the *PIG*'s ports inform a technology module that (1) the platform has collided, (2) the platform has received a beaming (it was found to lie within the region of a cone), or (3) the platform's equation of motion or expected lifetime is expired.

Technology modules obtain all other forms of environmental information from the SADMT/SF via procedure calls. Through this interface the technology modules can enquire and alter their physical properties (*e.g.*, mass, equation of motion, lifetime, and speed), emit cones, destroy themselves, and create new platforms (*e.g.*, fire a missile).

3.3.1. Represent Technology Modules in Ada

Since a TM is a SADMT process, the Ada representation of a TM is nearly identical to the Ada representation of a SADMT process. Both processes and technology modules must specify (1) the various types of data that flow on internal links, (2) the "semantics" of the process, (3) the subprocesses, (4) the ports, (5) the initialization for subprocesses and ports, and (6) the internal and inherited links. In addition, a technology module must specify the types of platforms that it may create, and the types of data that it may emit via a cone.

The generic packages *ConeDefiner_pkg* and *PlatformDefiner_pkg* define the procedures needed to create platforms and emit cones. These packages are located within the *Cones_n_Platforms* package. The *Cones_n_Platforms* package defines the procedural interface to the simulated environment. These procedures are used by the main program to initially create platforms to represent the SDS and the threat, and by the technology modules of each platform. They cannot be used by the logical processes of a platform.

Some of the key procedure and function are listed below. A complete specification is given in the *Cones_n_Platforms* package in Appendix D.

(1) *destroy_self* removes the platform from the simulation.

(2) *get_my_type* returns the platform's type.

(3) *change_my_type* alters the platform's type.

(4) *change_my_mass* alters the platform's mass.

(5) *get_physical_stuff* returns the *physical_stuff_block* record associated with the platform.

(6) *platform_position* returns the platform's current position.

(7) *platform_speed* returns the platform's current speed.

3.3.2. Dynamic Technology Modules

The SADMT system, as presented, requires that each platform specify all of its technology modules explicitly. This raises one important problem.

For example, sensor technology modules occur in pairs. For every type of sensor TM, there is a corresponding sensor response TM. For example, if a platform contains an XYZ sensor, then all other platforms which may be visible to an XYZ sensor must posses an XYZ sensor

response TM. Thus when a platform is "beamed" by an XYZ sensor cone, the XYZ sensor response module will return the appropriate echo.

The implication of this arrangement is that each time a new type of sensor module is employed, all of the platforms must be compiled with the corresponding (new) sensor response module. SADMT resolves this problem by providing a new kind of technology module called a dynamic technology module (DTM).

Dynamic technology modules (DTMs) are automatically placed on a platform when it is created. Therefore, a platform will contain one of each type of DTM not explicitly precluded by the platform.

A DTM is essentially identical to a TM with only one exception; a DTM can have at most three useful inputs port and no useful output ports. The DTM may have several input and output ports; however, only three of these ports can be linked. In particular, a DTM may link one port to each of the output ports of the parent platform's *PIG*. Figure 3-10 depicts a platform with the DTMs instantiated.

3.3.3. Representing Dynamic Technology Modules in Ada

A platform need know nothing about the presence of DTMs unless it chooses to exclude one or more of them. To exclude a DTM, the platform issues a call to the procedure *exclude_dyn_module*. This call specifies the *PDL_ptr* of the platform and a *PDL_string_ptr* to the name (designator) of the DTM to be excluded. Figure 3-11 shows a platform's *initialize* routine. In this example the platform excludes the dynamic technology module named *DYN_Gyro_name*.

There are a few differences in the Ada representation of a DTM as opposed to a TM. First, since DTMs are created by the simulation driver, the representation of a DTM must contain code to allow creation and initialization. The initialization is essentially identical to the normal TM except for the parameterization and the port specification. A DTM has no parameterization passed to it during the initialization since the initialization is not under the platform's control, and the port specification includes three *out* parameters used to tell the simulation driver which input ports to link to the *PIG*'s ports. There is one *out* parameter for each of the *PIG*'s ports. In Figure 3-12 only one of the *out* parameter is assigned a value. The *null* pointer is used for the other two parameters. As a result, only the *beamings* port of the *PIG* is linked in this example.

The creation of DTMs is similar to the creation of platforms. The primary difference is in the routines they call. A DTM package body (see Figure 3-12) calls *lookup_dyn_designator* while a platform's body calls *lookup_platform_designator.* The initialization procedures passed into the instantiation of *DTMCreator_pkg* and *PlatformCreator_pkg* packages also differ. The procedure passed into the *DTMCreator_pkg* (*e.g.*, in Figure 3-12 and Figure 3-13(a) the *initialize* function is passed into the *DTMCreator_pkg* package) receives the platform's *PDL_ptr* and returns up to three inports in the first three *out* parameters, while the procedure passed into the *PlaformCreator_pkg* receives the platform parameterization and returns the *PIG* in its *out* parameter.

3.4. Representing the Main Program in Ada

To simulate a system of platforms orbiting through space, a program to create the initial configuration of platforms and trigger the start of the simulation is needed. This program is referred to as the *main* program. The main program establishes the initial configuration of platforms by calling the procedure *create_platform*. After the configuration is established, the main program starts the simulation by calling the procedure *start_simulation*. The procedures *create_platform* are found in the instantiations of the *PlatformDefiner_pkg* located in each platform. As a result, the main program must "with" all of the packages representing platforms

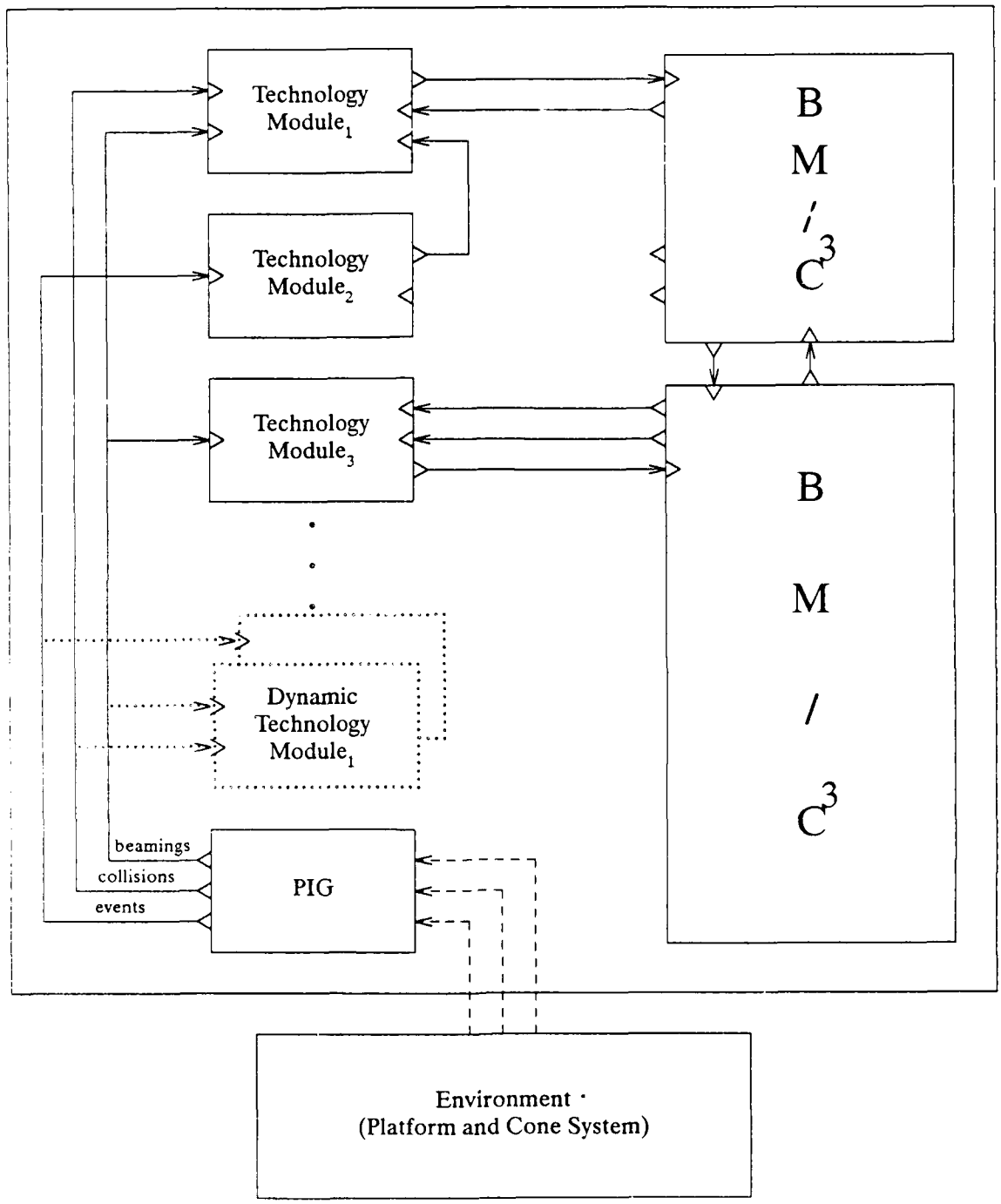


Figure 3-10 High-level View of a Platform

```
separate(Dyn_Platform_pkg)
procedure initialize(
            ZZ: out PIG_type;
            param: Dyn_Platform_parameterization_ptr;
            My_name: PDL_string_ptr:= Dyn_name;
            Discr_name: PDL_string_ptr:= empty_string;
            Characteristic: PDL_string_ptr:= empty_string) is
  Z: Dyn_Platform_type;
begin
  Z:= new Dyn_Platform_block;
  declare
    PIG: PIG_type renames Z.SUB.PIG;
    Parent_Process: Parent_Process_vector
   renames Z.SUB.Parent_Process;
    RW_Process: RW_Process_type renames Z.SUB.RW_Process;
  begin
    set_process_parent(Z.PDL, null, My_name, Discr_name, characteristic);
    initialize(PIG, Z.PDL);
    ZZ:= PIG;
    for i in 1 .. 3 loop
      initialize(Parent_Process(i), Z.PDL);
    end loop;

    internal_link(Parent_Process(1).message_out, Parent_Process(2).message_in);
    internal_link(Parent_Process(1).message_out, Parent_Process(3).message_in);
    internal_link(Parent_Process(2).message_out, RW_Process.message_in);
    internal_link(Parent_Process(3).message_out, RW_Process.message_in);
    internal_link(RW_Process.message_out, Parent_Process(1).message_in);
    exclude_dyn_module(Z.PDL, Dyn_Gyro_pkg.Dyn_Gyro_designator);
  end;
end initialize;
```

Figure 3-11. Platform Excluding a DTM

```
with Cones_n_Platforms;

package Dyn_Gyro_pkg is
  use Cones_n_Platforms;
  use PDL_pkg, Environ_Msg_pkg;

  type Dyn_Gyro_block;
  type Dyn_Gyro_type is access Dyn_Gyro_block;

  task type Dyn_Gyro_task is
    entry start_up(z: Dyn_Gyro_type);
  end Dyn_Gyro_task;
  type Dyn_Gyro_task_ptr is access Dyn_Gyro_task;

  type Dyn_Gyro_block is record
    PDL: PDL_ptr:= new_PDL_block(leaf);
    SEM: Dyn_Gyro_task_ptr;
    cone_in: Cone_Msg_ipptr:= new Cone_Msg_port(inport);
  end record;

  Dyn_Gyro_designator: constant PDL_string_ptr:= new string'("DYN_GYRO");
  Dyn_Gyro_name: constant PDL_string_ptr:= new string'("Dyn_Gyro");
  Dyn_Gyro_type_name: constant PDL_string_ptr:= new string'("Dyn_Gyro");
  Dyn_Gyro_discr_name: PDL_string_ptr:= empty_string;
  Dyn_Gyro_characteristic: constant PDL_string_ptr:=
   new string'("typename=Dyn_Gyro");

  procedure initialize(
              ZZcone: out Cone_Msg_ipptr;
              ZZevent: out Event_Msg_ipptr;
              ZZplatform: out Platform_Msg_ipptr;
              Parent: PDL_ptr);
end Dyn_Gyro_pkg;
```

Figure 3-12. Specification of a Dynamic Technology Module

```
with Vector_pkg;
package body Dyn_Gyro_pkg is
  use PDL_IO;
  use TXT_IO, INT_IO, TIME_IO, DURATION_IO, FLT_IO;
  use PDCone.Procedures;
  task body Dyn_Gyro_task is separate;
  procedure initialize(
              ZZcone: out Cone_Msg_ipptr;
              ZZevent: out Event_Msg_ipptr;
              ZZplatform: out Platform_Msg_ipptr;
              Parent: PDL_ptr) is separate;
  package Dyn_Gyro_Creator is
   new interface_procs.DTMCreator_pkg(
                    interface_procs.lookup_dyn_designator(Dyn_Gyro_designator),
                    initialize);
end Dyn_Gyro_pkg;
separate(Dyn_Gyro_pkg)
procedure initialize(
            ZZcone: out Cone_Msg_ipptr;
            ZZevent: out Event_Msg_ipptr;
            ZZplatform: out Platform_Msg_ipptr;
            Parent: PDL_ptr) is
  Z: Dyn_Gyro_type;
begin
  Z:= new Dyn_Gyro_block;
  declare
    cone_in: Cone_Msg_ipptr renames Z.cone_in;
    MYSELF: PDL_ptr renames Z.PDL;
  begin
    set_process_parent(Z.PDL, Parent, Dyn_Gyro_name, Dyn_Gyro_discr_name,
     Dyn_Gyro_characteristic);
    if init_debug_level > 150 then
      write_process_full(MYSELF, "*init> ", " before start_up");
    end if;
    initialize(Z.cone_in, Z.PDL);
    ZZcone:= Z.cone_in;
    ZZevent:= null;
    ZZplatform:= null;
  end;
  Z.SEM:= new Dyn_Gyro_task;
  Z.SEM.start_up(Z);

  if init_debug_level > 150 then
    write_process_full(Z.PDL, "*init> ", " after start_up");
  end if;
exception
  when others =>
    write_process_full(Z.PDL, "***Some error in ", "**");
end initialize;
```

Figure 3-13(a). Body of a Dynamic Technology Module

```
separate(Dyn_Gyro_pkg)

task body Dyn_Gyro_task is
  use Vector_pkg;
  use timing_ops;
  Z: Dyn_Gyro_type:= null;

  where: point_type;
begin
  accept start_up(Z: Dyn_Gyro_type) do
    Dyn_Gyro_task.Z:= Z;
    make_known(Z.PDL);
  end start_up;

  declare
    cone_in: Cone_Msg_ipptr renames Z.cone_in;
    MYSELF: PDL_ptr renames Z.PDL;
    package WAITING_pkg is new Wait_pkg(Z.PDL);
    use WAITING_pkg;
    package TM_INTERFACE_pkg is new Procedures_pkg(Z.PDL);
    use TM_INTERFACE_pkg;
  begin
    wait_for_initialization;
    loop
      where:= platform_position;
      write_process_full(MYSELF, "YOOOOOOOOOOOOOOOOOO ");
      put(where.x, 3, 3, 0);
      put(where.y, 3, 3, 0);
      put(where.z, 3, 3, 0);
      new_line;
      wait_for_activity(Time_out => 10);
      if Port_length(cone_in) /= 0 then
        write_process_id(MYSELF, "Cone received");
        consume(cone_in);
      end if;
    end loop;
  end;
end Dyn_Gyro_task;
```

Figure 3-13(b). Task Body of a Dynamic Technology Module

created in the initial configuration. Furthermore, the main program must "with" the *Cones_n_Platforms* package in order to access the data types and routines needed to establish the equation of motion for the individual platforms. This package also yields access to the *PDL_pkg* package. The *PDL_pkg* contains the time types and the IO packages.

Finally, the main program must "with" the *System_Simulation_pkg* package, given below:

```
with PDL_pkg;
package System_Scheduler is
  procedure start_simulation(stop_simulation_time: PDL_pkg.PDL_time_type:= 1500);
end System_Scheduler;
```

This package contains only one definition, the procedure *start_simulation*. After creating the initial configuration of platforms the main program issues the *start_simulation* procedure call, and specifies maximum amount of time the simulation will run. The time parameter is given as a *PDL_time_type*. This value can be converted to seconds, minutes, or hours using the constant *PDL_ticks_per_second*.

An example of a main program is given in Appendix E. In this example, the program (*main_sds0*) creates two *Command_Post_Platforms*, four *Sensor_Platforms*, several

Weapons_Platforms, and one *Russian_Missile_Base_Platform*. After creating these platform, this sample program calls the procedure *start_simulation* with a stop time of one *hour.*

3.5. Changes from the October Version

This section enumerates the changes to the Ada representation of the SADMT/SF interface since the October 16th draft of version 1.5. Readers who have not used the October draft should not concern themselves with these points.

(1) The *PlatformDefiner_pkg* package is a generic package which defines the *create_platform* procedure. The generic parameter is the parameterization type of the platform.

(2) The *PlatformCreator_pkg* package is a generic package that informs the simulation driver of the platform's existence and specifies the initialization procedure needed to properly establish this type of platform. The generic parameters are the parameterization type of the platform and the initialization procedure.

(3) The *ConeDefiner_pkg* package is a generic package which defines the *create_cone* procedures. The generic parameter is the data type to be transmitted over the cone.

(4) The *Procedures_pkg* package is a generic package which renames the procedures found in *interface_procs*. The generic parameter is the TM's *PDL_ptr.*

(5) The *start_simulation* procedure has been moved to *System_Scheduler* package.

(6) The *DONT_USE* package contains the procedure *notify_analog_part*. Never call this procedure!

(7) The method for defining the platform Parameterization (*PRM*) record has changed. The parameterization record is now visible. The declarations should be enclosed within a package.

(8) The parameters to the *set_process_parent* procedure have changed. This procedure now accepts two string pointers. The first describes the name of the object, and the second discriminates between several objects of the same name.

(9) The *cone_type* now includes a *source_point* and a *blackout_radius* field. The previous versions of SADMT/SF assumed the *source_point* to be the current position of the platform. Now, the TM must specify the source point of the cone. If the current position is needed, it can be acquired through the function *platform_position*. The *blackout_radius* field represent the radius of a sphere around the *source_point*, and no platform within this sphere will receive the cone. This change was made to provide a mechanism for realistic modeling.

(10) The *Cone_Msg* type now contains the field *cone_characteristic*. *Cone_characteristic* is of type *cone_type*. This represents the characteristics of the cone which caused the beaming.

(11) The creator task of platforms has been replaced by the generic package *PlatformCreator_pkg*. Consequently, access to the *master_creator* has been removed.

(12) The packages *Cone_Msg_pkg*, *Event_Msg_pkg*, and *Platform_Msg_pkg* have been added to SADMT. These packages group and rename the structures found in *Cones_n_Platforms.Environ_Msg_pkg* so they are compatible with the SAGEN naming conventions. (In the October release, these three packages were defined in the example of technology modules.)

CHAPTER 4

Behavioral Constraints

Luckham *et al.* have developed two powerful techniques, Anna [Luckham85] and TSL-1 [Luckham87], for adding assertions to an Ada program to capture behavioral constraints that are inappropriate to express in the language itself. One may think of Anna as specifying constraints on static program entities (types, variables, procedures, etc.) while TSL-1 allows one to specify constraints on the "event trace" of the executing program. While drawing heavily from these techniques and providing the same two basic kinds of constraints, the SADMT assertion facilities are far simpler because the SADMT model of execution has far fewer concepts. For example, since SADMT has no concept of a variable or of a procedure, there are no assertions in SADMT that deal with variables or procedures (or exceptions).[1] What SADMT does have is message types and ports; accordingly, there are assertions for constraining the "text" of message objects both across an entire message type and also for any particular input or output port. Other assertions deal with the maximum and minimum fan-in and fan-out of ports. As in TSL-1, a SADMT simulation may be viewed as a global stream of atomic events and one might like to constrain the sequence in various ways. For example, one might like to indicate the response time of a particular process by specifications like: "Within x time after the delivery of a message to some port, a message must be emitted from some other port." Note that each concept of this specification is a SADMT concept, not an Ada concept. SADMT provides facilities for these types of "atomic event trace" constraints. These mechanisms are much simpler than the corresponding TSL-1 mechanisms since the SADMT process model has far fewer events of interest.

While these annotations may be used for functional specification of process behavior, they are intended mainly for allowing run-time checks to be added automatically to the Ada program representing a SADMT specification. The primary function of ToolA mentioned in the Chapter 1 is to assist in the validation of these annotations by producing, for a given annotated program that represents a SADMT model, an equivalent Ada program that provides notification if any of the constraints given in the annotations are violated.

4.1. Annotations and Virtual Code

SADMT processes are represented in Ada using a specific structure of Ada packages, tasks and types. SADMT provides a facility for adding constaint information to the Ada program representing a SADMT model in the form of structured comments. There are two types of SADMT structured comments: **annotations** and **virtual code**. Syntactically, each line of a SADMT structured comment must begin with the characters "--:". (Consequently, other Ada comments must not begin with "--:" or they will be treated as SADMT structured comments.)

The first type of SADMT structured comments, **annotations**, may be used to provide information about the model that is not reasonably represented directly in Ada. For example, SADMT provides an annotation for asserting what constraints must hold on the text of all messages emitted from a certain port. It has been argued that such port constaints may be adequately represented simply by inserting *if*-statements around the emits for that port to test the

[1] Of course, when one writes the semantics of a process in Ada, one will obviously use Ada procedures and variables and one might desire to annotate this code using an annotation language for Ada, *e.g.*, Anna. But, these procedures and variables are part of the implementation and not part of the SADMT model *per se*.

appropriate conditions. However, the Ada program realizing the semantics may use a number of procedures with port-typed parameters; in such a case, finding the correct emits may be impossible. Also, such constraints should appear in the part of the program that represents the SADMT specification rather than in the part that represents SADMT process semantics. Instead, what is really needed is to have a tool that automatically inserts code to test assertions and also inserts the appropriate changes in the simulation driver to implement the tests in terms of the process model. Nevertheless, it is still meaningful to execute the program without the testing code but no indication of assertion violations would be given.

The second type of SADMT structured comment is **virtual code**. Virtual code is used for describing entities in Ada that are not actually part of the underlying Ada program but can be readily represented using Ada constructs. In the current version of SADMT, virtual code is used almost exclusively to specify the predicates needed in the annotations, *i.e.*, Ada programming is used as a base mathematical semantics; thus, Ada Boolean expressions are used as propositions of predicate calculus and Ada loops and searches are used to replace quantifiers. While virtual code is not distiguishable from "normal Ada text it should be treated specially because it is code that would not be executed to realize the semantics of the underlying SADMT model. In other words, the sole reason for the existence of virtual code is to define entities referenced in annotations.

4.2. SADMT Structure Names

A unique name is associated with each "block" of SADMT structured comments. Syntactically, the name is an arbitrary string of printable characters not containing a colon or a vertical bar("|"). ToolA will provide the capability for selectively enabling and disabling SADMT structure blocks by referring to the structure name. Thus, the architect may choose among various options trading off thoroughness of constraint checking for increases in compilation or simulation speed. Also, early implementations of process semantics may not satisfy all of the constraints that the final system will satisfy; thus, it is important to be able to suppress the constraint checking during early development.

The function of ToolA is actually two-fold. ToolA is responsible for translating annotations into Ada code that is used to check specified constraints. Since this code is not part of the execution semantics of the model, this output of the translation is in virtual code. Second, ToolA is responsible for turning virtual code into physical code according specifications that are supplied by the user. Specifically, ToolA accepts from the user a description of what virtual code is to be made physical in terms of SADMT structure names. It is the user's responsibilty to ensure that the resulting program is a correct Ada program; this is normally accomplished by coordinating the structure name for an annotation with the structure names of the virtual code blocks that are needed for the annotation.

4.3. Annotations

The syntax of a SADMT annotation is

--:[<str-name>]| <keyword> <text> ;

where

<str-name> is the SADMT structure name,

<keyword> is one of the SADMT annotation keywords, *e.g.*, **message_constraint** or **define_event**, and

<text> is whatever parameters are meaningful for the particular keyword specified, but no semicolons. An example of annotations is found in Figure 4-1 and Figure 4-2.

A multiline format for annotations is also available:

--:[<str-name>]| <keyword> <text1>

```
--:| <text2>
--:| <text3>
 ...
--:| <textn> ;
```

The intention here is that <text1>, <text2>, ... <textn> are concatenated (separated by spaces) to obtain the <text> of the single line form. (Again, note that the <text> does not contain any semicolons; thus, there is no ambiguity.)

In the text below, a number of different annotations are defined. In each case, constraints are specified for where each annotation may appear in the Ada text representing a SADMT model. In many cases, an annotation processor will be required to replace an annotation by an Ada fragment declaring some objects and another Ada fragment declaring code for these objects. In such cases, an annotation processor may place additional constraints on the Ada compilation structure of the model. For example, it may require that the package specification for a SADMT process and the package body for that process be in the same compilation unit (or file).

4.4. Virtual Code

Virtual code is a mechanism for adding Ada code to a design that provides additional insight or explanation and yet is not required in specifying the functionality or design of a process. This approach is a simplified version of that taken in Anna [Luckham and Henke 85]. The general syntax of a line of virtual code is

```
--:[<str-name>]:<Ada code>
```

where

<str-name> is the structure name of this SADMT structure, and

<Ada code> is some portion of a valid Ada program.

A line of virtual code for which the optional name is not explicitly indicated is treated as though it has the same name as the last name explicitly indicated. Thus, there is no need for a multiline syntax.

A valid Ada program must result when the enabled virtual code is added to the Ada program that contains it. The virtual code must not alter variables of the original Ada program. Thus (as in Anna), virtual code has "read only" access to variables of the original program. (There are additional constraints that must be satisfied to ensure that the virtual code has absolutely no effect on the "underlying Ada program"; *e.g.*, no heap variables may be declared, no extra variables may be declared in a procedure's activation record, etc. Anna defines a set of constraints that may be checked at compile time and that ensures that virtual code does not affect the data space of a program. Of course, the presence of the virtual code in the program module may affect the paging behaviour of the underlying program.) Virtual code may, however, declare and modify its own variables. In this way, virtual code may be used to "implement" sequential constraint checks as well as combinational ones.

4.5. Assertions for Ports and Message Types

A number of different types of annotations are provided to assert constraints on message types and port structure:

(1) message constraints,

(2) process message constraints,

(3) port message constraints,

(4) port structure and buffering constraints.

(5) redundant linkage constraints.

Each of these is discussed individually in subsequent sections. An example program is given in Figure 4-1 and Figure 4-2. Here, we shall only summarize what each type of constraint specifies. The first three constraints assert the conditions on the contents of messages. The difference is in the placement of the assertions and the "coverage." For example, a message constraint is specified in the same package that defines the port types for a message type. Such a constraint applies to every port that uses the defined port types. A process message constraint is placed in the process specification and specifies that the constraint applies to ports (of the given port types) that are defined in (or inherited from) the given process. A port message constraint follows the initialization of the port and refers only to the port specified.

A port structure or buffering constraint places constraints on the maximum and minimum fan-in or fan-out of a process; it is also possible to constrain the number of actual links. In addition, these constraints may specify the maximum number of messages that may be queued at an input port. A redundant linkage constraint allows a specification of what is **not** connected. For complicated linkage patterns, these annotations provide some protection against typographical errors. A detailed discussion follows.

4.5.1. Message Constraints

The format of a message constraint annotation is

```
--:| message_constraint <porttype> <predicate>;
```

The annotation must appear in the same package with the instantiation of the *PortDefiner_pkg* package for a message type *T*. The <predicate> specified takes a single argument of type *T*; the <porttype> is the type defined in the instantiation of *PortDefiner_pkg* for ports on type *T*, *i.e.*, *PD.T_port*. Since *PD.T_port* must be known, the annotation must follow the instantiation of the *PortDefiner_pkg* package. Further, it must appear at a point in the code where it could be replaced by the declaration[2]

```
package NEWNAME is
  use PD.Procedures;
  port: <porttype>;
  b: Boolean:= <predicate>(Base_Type(port.all));
end NEWNAME;
```

resulting in a legal Ada program, where NEWNAME is a name that is otherwise free[3] in this package.

As mentioned previously, this annotation specifies that all ports of the message type are checked by the constraint. The package in Figure 4-1 constrains a *Simple_Msg* so that the *time_created* is always greater than or equal to zero.

---

[2] The following functions are defined in the *PortDefiner_pkg* package (see Appendix A): *Base_Type*, *Ensure_Port*, *Ensure_Output_Port*, *Ensure_Output_Port*, *Ensure_Input_Port*, and *Ensure_Input_Port*. There are no semantics associated with them; they are only used in this chapter to help identify what annotations are legal.

[3] For the purposes of this paper, one could define a name as **free** if it did not appear elsewhere in the program, *i.e.*, if it is a **new** name with respect to the program. A more formal definition of the term would simply account for its "declarability" at any particular place in the program.

```
with PDL_pkg, PortDefiner_pkg;
package Smpl_Msg_pkg is
  type route_type is array(1 .. 20) of integer;
  type Simple_Msg is record
    time_created: PDL_pkg.PDL_time_type;
    route: route_type;
    last_slot: integer:= 0;
  end record;
  Simple_msg_Debug_Class: string(1 .. 10):= "Simple_msg";
  procedure put_msg(m: Simple_Msg; indent: integer:= 20);

  package PD is
      new PortDefiner_pkg(
                Simple_Msg,
                put_msg,
                Simple_msg_Debug_Class);
  subtype Simple_Msg_port is PD.T_port;
  subtype Simple_Msg_ipptr is PD.T_ipptr;
  subtype Simple_Msg_opptr is PD.T_opptr;

--:: function valid_time(msg: Simple_Msg) return boolean;

--:| message_constraint Simple_Msg_port
--:|     valid_time;

end Smpl_Msg_pkg;

package body Smpl_Msg_pkg is

--:: function valid_time(msg: Simple_Msg) return boolean is
--::   use PDL_pkg.timing_ops;
--:: begin
--::   return msg.time_created >= 0;
--:: end valid_time;

  procedure put_msg(m: Simple_Msg; indent: integer:= 20) is
    use PDL_pkg.PDL_IO;
    use txt_io, int_io, time_io;
  begin
    for i in 1 .. indent loop
      put(' ');
    end loop;
    put("the message ts,r=");
    put(m.time_created, 1);
    put(":");
    for i in 1 .. m.last_slot loop
      put(m.route(i), 1);
      put(":");
    end loop;
    new_line;
  end put_msg;
end Smpl_Msg_pkg;
```

Figure 4-1. Message Constraints

4.5.2. Process Message Constraints

The format of a process message constraint annotation is

```
--:| process_message_constraint <porttype> <predicate>;
```

The annotation must appear in the initialization procedure for the process **after** the call to *set_process_parent* but **before** the initialization of any port that uses the port type. Further, it must appear at a point in the code where it could be replaced by the block

```
declare
— assume that the use PD.Procedures for this porttype
— has already been elaborated
   port: <porttype>;
   b: Boolean:= <predicate>(Base_Type(port.all));
begin
   null;
end;
```

resulting in a legal Ada program. This annotation specifies that all ports of the message type that are defined in or inherited from the given process are checked by the constraint. Figure 4-2 demonstrates a **process_message_constraint**. This constraint insures that the *last_slot* field of a *Simple_Msg* is always greater than or equal to zero.

4.5.3. Port Message Constraints

The format of a port message constraint annotation is

```
--:| port_message_constraint <port> <predicate>;
```

The annotation must appear in the initialization procedure for the process defining the named port after the initialization of that port. Further, it must appear at a point in the code where it could be replaced by the block

```
declare
— assume that the use PD.Procedures for this porttype
— has already been elaborated
   b: Boolean:= Ensure_Port(<port>.all) and
     <predicate>(Base_Type(<port>.all));
begin
   null;
end;
```

resulting in a legal Ada program. This annotation specifies that all messages on the named port are checked by the constraint. In Figure 4-2 a **port_message_constraint** is used to check that the *time_created mod 40* is equal to zero for the output port *Z.message_out*.

```
package body Prnt_Process_pkg is
  use PDL_IO;
  use TXT_IO, INT_IO, TIME_IO, DURATION_IO;
  use Smpl_Msg_pkg.PD.Procedures;

  procedure initialize(
            Z: in out Parent_Process_type;
            Parent: PDL_ptr;
            My_name: PDL_string_ptr:= Parent_process_name;
            Discr_name: PDL_string_ptr:= empty_string;
            Process_type_name: PDL_string_ptr:= Parent_process_type_name;
            Characteristics: PDL_string_ptr:= Parent_process_characteristic) is
    wt1: constant PDL_duration_type:= 20;
    wt2: constant PDL_duration_type:= 30;
    wt3: constant PDL_duration_type:= 50;
    discr_n1: constant PDL_string_ptr:= new string'("1");
    discr_n2: constant PDL_string_ptr:= new string'("2");
    discr_n3: constant PDL_string_ptr:= new string'("3");

--::    function valid_slot(msg: Simple_Msg) return boolean is
--::    begin
--::      return msg.last_slot >= 0;
--::    end valid_slot;

--::    function time_mod(msg: Simple_Msg) return boolean is
--::      use PDL_pkg.timing_ops;
--::    begin
--::      return ((msg.time_created - 0) mod 40) = 0;
--::    end time_mod;

--::    function Q_length_limit(msg: Simple_Msg) return integer is
--::      use PDL_pkg.timing_ops;
--::    begin
--::      return 20;
--::    end Q_length_limit;

  begin
    Z:= new Parent_Process_block;
    set_process_parent(Z.PDL, Parent, My_name, Discr_name);
--:|      process_message_constraint Simple_Msg_port
--:|        valid_slot;

    initialize(Z.SUB.P1, Z.PDL, Discr_name => discr_n1, waittime => wt1);
    initialize(Z.SUB.P2, Z.PDL, Discr_name => discr_n2, waittime => wt2);
    initialize(Z.SUB.P3, Z.PDL, Discr_name => discr_n3, waittime => wt3);

    initialize(Z.message_in, Z.PDL);

--:|      port_constraint Z.message_in
--:|        Q_length_limit;

    initialize(Z.message_out, Z.PDL);

--:|      port_message_constraint Z.message_out
--:|        time_mod;

    inherited_link(Z.message_in, Z.SUB.P2.message_in);
    inherited_link(Z.SUB.P3.message_out, Z.message_out);
    internal_link(Z.SUB.P2.message_out, Z.SUB.P1.message_in);
    internal_link(Z.SUB.P2.message_out, Z.SUB.P3.message_in);
--:|      linkage_constraint
--:|        Z.SUB.P1 Z.SUB.P2;

    make_known(Z.PDL);
  exception
    when others =>
      Put_line("**Some error in PARENT_init**");
  end initialize;
end Prnt_Process_pkg;
```

Figure 4-2. Other Constraints

4.5.4. Port Structure and Buffering Constraints

The format of a port constraint annotation is

--:| **port_constraint** [<port>] <attributes>{,<attributes>};

where <attributes>:=

max_fanin => <integer_expression> |
max_fanout => <integer_expression> |
min_fanin => <integer_expression> |
min_fanout => <integer_expression> |
max_linksin => <integer_expression> |
max_linksout => <integer_expression> |
min_linksin => <integer_expression> |
min_linksout => <integer_expression> |
max_Qlength => <integer_expression>

If no port is named, then the effect of this annotation is that the attributes specified become the default for any port of some process that is not yet initialized. Thus, it appears in the initialization procedure for that process after the call to *set_process_parent*. If a port is named, the annotation must appear in the initialization procedure for the process defining the named port after the initialization of that port. In both cases, it must appear at a point in the code where it could be replaced by the block

```
declare
-- assume that the use PD.Procedures for this porttype
-- has already been elaborated
   b: Boolean:= Ensure_Port(<port>.all);
   max_fanin: integer:= <integer_expression>;
   max_fanout: integer:= <integer_expression>;
   min_fanin: integer:= <integer_expression>;
   min_fanout: integer:= <integer_expression>;
   max_linksin: integer:= <integer_expression>;
   max_linksout: integer:= <integer_expression>;
   min_linksin: integer:= <integer_expression>;
   min_linksout: integer:= <integer_expression>;
   max_Qlength: integer:= <integer_expression>;
begin
   null;
end;
```

resulting in a legal Ada program, where the integer declarations correspond directly to those specified in the annotation, and if no port is named then the "*b:Boolean...;*" is omitted.

Again, this annotation either specifies characteristics of some port that are to be checked, or it specifies defaults that are to be checked for some subset of the ports in some process. The checking occurs during process initialization, except for **max_Qlength**. **Max_QLength** is checked dynamically. Figure 4-2 illustrates a **port_constraint**. This constraint limits the queue length of the input port *Z.message_in* to twenty.

4.5.5. Redundant Linkage Constraints

The format of a redundant linkage constraint annotation is one of the following

```
--:| linkage_constraint <output_port> <input_port>;
--:| linkage_constraint <output_port> <process_B>;
--:| linkage_constraint <process_A> <input_port>;
--:| linkage_constraint <process_A> <process_B>;
```

Redundant linkage constraint annotations must appear in the initialization procedure for a process after the initialization of ports of that process. Further, it must appear at a point in the code where it could be replaced by the block

```
declare
--- assume that the use PD.Procedures for this porttype
--- has already been elaborated
   b1: Boolean:= Ensure_Output_Port(<output_port>);
   b2: Boolean:= Ensure_Input_Port(<input_port>);
   b3: Boolean:= Ensure_Process(<process_a>.PDL);
   b4: Boolean:= Ensure_Process(<process_b>.PDL);
begin
   null;
end;
```

resulting in a legal Ada program, where the declaration of b1, b2, b3 and b4 is omitted if the annotation does not name <output_port>, <input_port>, <process_A>, and <process_B>, respectively. For the first case, the annotation specifies that the <output_port> is **not** connected to the <input_port>. For the second form, the annotation specifies that the <output_port> is **not** connected to **any** input port of <process_B>. For the third form, the annotation specifies that **no** output port of <process_A> is connected to the <input_port>. Finally, the last form of the annotation specifies that **no** output port of <process_A> is connected to **any** input port of <process_B>.

The information specified by these annotations is completely redundant since the linkage is completely specified. These annotations simply provide extra protection against simple typographical errors that may occur when the linkage patterns are relatively complicated.

4.6. Assertions on the Atomic Event Trace

All of the previous constraints are constraints that apply at a point in time. There are other types of constraints in which it is necessary to consider behaviour over a window in time. For example, a response constraint might specify that some atomic event, perhaps the emission of a certain kind of message from some port, must occur within, say, forty milliseconds after the receipt of some second kind of message at some other port. Such a constraint links multiple instants in time and cannot, therefore, be established by the same mechanism as the constraints in the last section.

The new notion that underlies these "window-in-time" constraints is the concept of an atomic event trace, that is, a sequence of timestamped atomic events that transpire during program execution. First, mechanisms must be established for defining the atomic event trace. Second, mechanisms must be established for specifying constraints on the atomic event trace. These are the topics of the next two sections.

4.6.1. Defining Atomic Events

The atomic event trace is defined for SADMT in terms of visible phenomena of the SADMT process model, specifically, the receipt of messages at ports and the emission of messages from ports. In order to simplify the discussion, we say that an incoming message *interacts* with an input port when it is received and that an outgoing message *interacts* with an output port when it is emitted. In actual fact, the set of all such interactions is a superset of the events of interest and attempting to track all such events is not practical. What is needed is to be able to specify deliveries and emissions that are events of interest. This is done by using a **define_event** annotation, the syntax of which is the following.

```
--:| define_event      <event_name>
--:|                   <port_name_skeleton>
--:|                   [//<predicate>[:<selector>(<selector_type>)]]
--:|                   {,<variable> in <range>};
```

Before proceeding to a detailed explanation of the various components of an atomic event definition, one or two examples may be helpful. In the following, assume that

(1) the messages in this example are of type *integer*,

(2) *a* is a process with two ports *a1* and *a2*,

(3) *b* is a one-dimensional array of processes, each of which has a port *b1*,

(4) *c* is a two-dimensional array of processes, each of which has a one-dimensional array of ports *c1*

Then, the following might be a set of atomic event definitions.

```
--:: function even(x: integer) return Boolean is
--:: begin
--::   return (x mod 2) = 0;
--:: end even;
--:: function negative(x: integer) return Boolean is
--:: begin
--::   return x < 0;
--:: end negative;

--:| define_event event_1 a.a1//even;
--:| define_event event_2 a.a2//negative;
--:| define_event event_3 b(i).b1//even, i in 'A'..'Q';
--:| define_event event_4 c(i,j).c1(k)
--:|             , i in c'range(1)
--:|             , j in c'range(2)
--:|             , k in c1'range;
```

These annotations define atomic events as follows.

(1) The first annotation specifies that *event_1* occurs whenever a message interacts with *a.a1* and the content of the message satisfies the predicate *even*.

(2) Similarly, *event_2* occurs whenever a message interacts with *a.a2* and the content of the message satisfies the predicate *negative*.

(3) The next annotation actually defines an array of atomic events, *event3('A'..'Q')*. In particular, for each *q* in *'A'..'Q'*, *event_3(q)* occurs whenever a message interacts with *b(q).b1* and

the content of the message satisfies the predicate *even*.

(4) The last annotation defines a three-dimensional array of atomic events. Specifically, *event_4(x1,x2,x3)* occurs whenever a message interacts with *c(x1,x2).c1(x3)*. Since no predicate is specified, the atomic event occurs regardless of the content of the message.

Returning to a more rigorous discussion of the components of an atomic event definition, each optional phrase "<variable> in <range>" is called a **quantifier**. If quantifiers are not present then a single scalar atomic event is defined; otherwise, an array of atomic events is defined whose dimension is the same as the number of quantifiers. For an array atomic event definition, the dimensions are typed according to the ranges specified in the order specified. As usual, Ada scoping must be obeyed; that is, if the annotation is replaced by

```
<event_name>_object: PDL_pkg.Event_pkg.regular_expression;
function <event_name>(P: PDL_ptr)
   return PDL_pkg.Event_pkg.regular_expression;
```

or

```
subtype <param_type-1> is <range-1>;
subtype <param_type-2> is <range-2>;
   .
   .
type <array_type> is array(<param_type-1>, <param_type-2>,
 ...) of PDL_pkg.Event_pkg.regular_expression;
<event_name>_object: <array_type>;
function <event_name>(
            <variable-1>: <param_type-1>;
            <variable-2>: <param_type-2>;
            ...
            P: PDL_ptr)
   return PDL_pkg.Event_pkg.regular_expression;
```

(depending or the scalar- or array-ness of the definition) then a legal Ada program must result. Here, each of <param_type-k> and also <array_type> is a name that is free at this point in the program. The function declaration and associated <param_type> declarations, if any, are actually placed into the Ada code by the annotation processing tool, *i.e.*, the <event_name> function declaration is actually elaborated at the point of the annotation. Conversely, the object declarations here simply indicate that the <event_name>s must be free; these declaration are likely not placed into the actual Ada code.

In addition to the function declaration, other declarations and code must be inserted into the Ada representation of the SADMT model to declare and initialize the objects used by the simulation driver to actually implement capturing and constraining the event trace. The code about to be described is useful for determining the legality of an annotation; however, it is **not** the actual code that would be used by an annotation processor.

The <port_name_skeleton> is an Ada <name> that yields a port *P* when the quantified variables of the atomic event definition receive a value in the specified range. Specifically, *P* is the port associated with the event *<event_name>(var-1,var-2,...)*. Note that if the <port_name_skeleton> uses fewer subscripts than there are quantifiers, then the same port will be associated with more than one event. This can be useful if the <selector> form of <predicate> is specified.

Finally, the <predicate>, if specified, determines whether the specified atomic event actually occurs, given that a message has interacted with the specified port. The <predicate> is a function of one or two parameters that returns a Boolean result. The type of the first parameter of the <predicate> is the message type of the port. A one-place predicate is sufficient for all scalar cases and the overwhelming majority of array cases. However, sometimes the determination of whether an atomic event is of interest depends both on the contents of the message **and** on which port has the interaction. In order that the user not have to specify different predicates for each element of an atomic event array, a <selector> is used. The type of the second argument to the <predicate> is then the same as the result type <selector_type> of the <selector>.

Given this, the following must be legal as the code corresponding to the **define_event** annotation if the code were placed in the package body for a process just before the elaboration of the procedure body for *initialize*.

```
package body NEWNAME is
  b: Boolean;
  seelect: <selector_type>;
begin
  for <variable-1> in <range-1> loop
    for <variable-2> in <range-2> loop
          .
          .
      seelect:= <selector>(<variable-1>, <variable-2>, ...);
      b:= <predicate>(Base_type(<port_name_skeleton>.all), seelect);
          .
          .
    end loop;
  end loop;
end NEWNAME;
```

For a scalar definition, there would be no loops or any constructs dealing with the <selector>. Obviously, this is **not** the code that is actually generated.

There is a choice for the placement of **define_event** annotations depending on whether the event functions are to be visible externally or not. If the event functions are to be visible externally, then the annotations should be placed in the public part of the package defining the process in which the events are defined; otherwise, the annotations should be placed into the <basic_declaration> part of the package body. The annotation processor is responsible for adding appropriate function bodies and for adding code to the "initialization" procedure for defining and initializing the objects needed by the driver. **Importantly,** the annotation processor is also responsible for renaming each event function so that the last parameter defaults to specify the process containing the **define_event** annotation.

4.6.2. Constraining the Atomic Event Trace - Syntax

In the previous section, the concept of SADMT atomic events is defined. In this section, the format of constraints that may be specified against the atomic event trace is discussed. For reasons of computational efficiency, the types of constraints that one may specify is substantially less than all possible constraints that one might imagine. However, the intention is that the type of constraint actually encountered in system specifications can be specified in a straightforward way.

The atomic event trace is a sequence of pairs where

(1) the first element of the pair is the atomic event.

(2) the second element of the pair is the time when the atomic event occurred.

One may view the atomic event trace as a string of atomic events (ignoring the timestamps); what is needed is some way to specify when something interesting has happened. Some events of interest are not atomic but are composed of having several atomic events occur in a given sequence. Thus, interest is aroused when a suffix of the event trace has some particular property. In the case of SADMT sequence constraints, the suffixes of interest are defined by means of regular grammars, in particular, by regular expressions.

A regular expression can be formed from the following rules:

(1) An atomic event is a regular expression denoting the set of strings of unit length whose only symbol has that event as the first element of the pair.

(2) The expression *empty_string* is a regular expression denoting the set of strings whose only member is the empty string.

(3) The expression *any_event* is a regular expression denoting the set of strings of unit length.

(4) If A is a regular expression, then *not* A is a regular expression denoting the complement of the set of strings denoted by A.

(5) If A and B are regular expressions, then $A + B$ is a regular expression denoting the set of strings formed by concatenating an element of the set denoted by A with an element of the set denoted by B.

(6) If A and B are regular expressions, then A *and* B is a regular expression denoting the intersection of the sets denoted by A and B.

(7) If A and B are regular expressions, then A *or* B is a regular expression denoting the union of the sets denoted by A and B.

(8) If A is a regular expression, then *closure(A)* is a regular expression denoting the set of strings formed by selecting an arbitrary number of strings from the set denoted by A and concatenating them.

The above rules are the usual ones for forming regular expressions; however, a number of regular sets are hard to specify in this way. Thus, a number of extra specification concepts are available.

(9) If AA is a sequence (*i.e.*, a one-dimensional array) of regular expressions, then *any(AA)* is the union over the sequence of the sets denoted by the elements of AA.

(10) If AA is a sequence of regular expressions, then *all(AA)* is the intersection over the sequence of the sets denoted by the elements of AA.

(11) If AA is a length-n sequence of regular expressions, then *each(AA)* denotes the set of strings formed in the following manner:

a) select one string each from the set denoting by the elements of AA.

b) select (n-1) arbitrary strings (*i.e.*, from *closure(any_event)*), called glue strings.

c) the string is formed by concatenating these ($2n$-1) strings in an arbitrary order so long as glue strings are alternated with nonglue strings.

Thus, each string in *each(AA)* contains a string from each of the denoted sets in an arbitrary order (but non-overlapping) and separated by arbitrary "glue."

(12) If AA is a sequence of regular expressions, then the strings of *sequence(AA)* are formed in the same way as *each(AA)* except that the nonglue strings appear in the same order in the composed string as the component regular expressions appear in AA.

(13) As a special case of the above, if A and B are regular expressions, then $A > B$ is the same as $((A,B))$. For example, a string in $A > B$ is a string from the set denoted by A followed by

an arbitrary string followed by a string from the set denoted by *B*. The construction "*(A,B)*" is an Ada construct representing a two-element array literal.

(14) Lastly, no string represents a regular expression unless it is constructed according to these rules. Note, the Ada precedence rules are used to disambiguate all constructions.

We may now proceed to the definition of a sequence constraint annotation, the syntax of which is the following.

```
--:|   sequence_constraint
--:|         if
--:|                 <trigger_RE_skeleton>
--:|         then [not]
--:|                 <consequent_RE_skeleton>
--:|                  [      before <race_RE_skeleton>
--:|                   | within <PDL_duration_expression>]
--:|         { "|"   <consequent_RE_skeleton>
--:|                  [      before <race_RE_skeleton>
--:|                   | within <PDL_duration_expression>]
--:|         {,<variable> in <range>};
```

A sequence_constraint annotation is legal only if replacing the annotation by

```
declare
  RE: PDL_pkg.Event_pkg.regular_expression;
  dur: PDL_duration_type;
begin
  for <variable-1> in <range-1> loop
    for <variable-2> in <range-2> loop
          .
          .
      RE:= <trigger_RE_skeleton>;
      RE:= <consequent_RE_skeleton-1>;
      RE:= <consequent_RE_skeleton-2>;
            .
            .
      RE:= <race_RE_skeleton-1>;
      RE:= <race_RE_skeleton-2>;
            .
            .
      dur:= <PDL_duration_expression-1>;
      dur:= <PDL_duration_expression-2>;
            .
            .
          .
          .
    end loop;
  end loop;
end;
```

results in a legal Ada program. Normally, sequence constraint annotations must appear in the "initialization" procedure of a SADMT process. They are placed immediately after the port

linkage section of the initialization. The exception is that sequence constraint annotations are placed in the code in those sections where new platforms are created and where it is desired to relate events on one platform to events on another. In such a case, the atomic events used must be those that are visible in the specification of the platform; clearly, the *PDL_ptr* must be supplied and the full form of the "event functions" used in these cases. Importantly, this placement of **sequence_constraint** annotations allows (and implies) that sequencing constraints specified at nonleaf nodes are enforced even though the semantics of nonleaf nodes is not executed.

4.6.3. Constraining the Atomic Event Trace - Semantics

There are two parts to the semantics of a **sequence_constraint**. The first is to determine when the constraint is "triggered," *i.e.*, when it applies. The second part is to determine what property the atomic event trace must satisfy so that the constraint is also satisfied.

Each sequence constraint *C* has a component, *C.trigger_RE*, that is called the triggering regular expression for *C* and this regular expression determines when the constraint is triggered. Specifically, if RE is a regular expression, then a "triggering suffix for RE" is the shortest suffix *S* of the atomic event trace so that *S* matches RE and no prefix of *S* matches RE. A sequence constraint is triggered if there is a triggering suffix for *C.trigger_RE*.

Assume for the moment that **not** is not specified following the **then** in sequencing constraint *C*. Then, *C* is satisfied iff **neither** of the following conditions holds:

(1) one of the performance periods expires before a triggering suffix *S'* occurs for the corresponding *consequent_RE*, that is, no such *S'* has occurred where the latest event in *S'* occurs after the latest event in *S* and before the time of the **earliest** event in *S* plus the period specified (infinite duration is the default).

(2) one of the *race_RE* triggers before the corresponding *consequent_RE* triggers.

Conversely, if **then not** is specified, then *C* is satisfied only if

(1) **no** *consequent_RE* has a triggering suffix within the specified period.

(2) **no** *consequent_RE* triggers before the corresponding *race_RE*.

One final consideration is that any particular atomic event can be used as the last symbol of a *consequent_RE* triggering suffix to retire at most one instance of the same sequencing constraint. However, the same atomic event could be the last symbol in the triggering suffix used to retire instances of different sequencing constraints.

In the following example, *event_A* will trigger the constraint.

```
--:|  sequence_constraint
--:|  if  event_A
--:|  then
--:|       event_B > each(event_C, event_D)
--:|            before event_E;
```

Once the constraint is triggered, then an occurrence of *event_B* followed by both *event_C* and *event_D* must occur before *event_E*. Thus, the following event trace, *event_A*, *event_Q*, *event_B*, *event_Z*, *event_D*, *event_X*, *event_C*, would satisfy the constraint as would the trace *event_A*, *event_B*, *event_X*, *event_C*, *event_Z*, *event_D*.

The following event trace will trigger and satisfy the example constraint twice: *event_A*, *event_Q*, *event_A*, *event_Z*, *event_B*, *event_R*, *event_D*, *event_C*, *event_C event_X*, *event_E*. In this trace, *event_A*, triggers the constraint twice, and event sequence from *event_Z* through the first occurrence of *event_C* satisfies the second triggering, and the next occurrence of *event_C*

satisfies the first triggering.

Nevertheless, the following event trace will violate the sequence constraint: *event_A*, *event_Q*, *event_A*, *event_B*, *event_R*, *event_D*, *event_C*, *event_X*, *event_E*. The event trace triggers the sequence twice, once for each occurrence of *event_A*; however, the constraint is only satisfied once by *event_C*. In particular, *event_C* is may be used to satisfy (retire) at most one instance of the same sequencing constraint. Therefore, the second triggering of the sequence constraint is satisfied by this event trace, while the first triggering is waiting for an occurrence of *event_C* when *event_E* occurs.

In the following example, a similar constraint is constructed; however, multiple occurrences of *event_A* will only trigger the constraint once provided they occur before *event_B*.

```
--:|    sequence_constraint
--:|    if  event_A
--:|    then
--:|        closure(not(B))+event_A
--:|      | event_B > each(event_C, event_D)
--:|            before event_E;
```

In this example, any number of *event_A*s may occur before *event_B*. Therefore, the trace (given above) that violated the previous constraint will satisfy this constraint. Once again, the event trace triggers the sequence twice, once for each occurrence of *event_A*; however, the second triggering also satisfies and retires the first triggering of the constraint.

CHAPTER 5

Simulating the Effect of the Execution Environment

The assumptions that an architect makes about the underlying environment may be fundamental to the specification of the architecture. Such assumptions may be modeled directly in the process semantics. However, it is usually more desirable to specify environmental effects separately. It is particularly desirable to be able to make changes in the environmental assumptions without making any changes in the text of the process semantics. SADMT provides mechanisms to model the effect of the execution environment. Slight changes in the process semantics may be required when the semantics are first coded in order to make full use of these mechanisms.

There are two possible interactions allowed between the execution environment and the specification of the architecture. First, SADMT provides a mechanism by which a possibly nonzero, possibly random **transit delay** can be specified for internal links. In this way, simple probabilistic assumptions about interprocess communications can be modeled easily. There are situations in which this simple probabilistic approach is not appropriate; in such cases, a user may have to introduce a SADMT process to explicitly represent the behaviour of the communications being modeled. Second, SADMT provides a mechanism whereby certain of the "wait"s issued by a SADMT process can be externally controlled, that is, controlled by a module other than the process semantics. This mechanism offers highly flexible control over **processing delays**, but at the cost of some verbosity. As will be seen, it is substantially simpler to model simple probabilistic assumptions than to model detailed resource contention.

An important driver in the design of these SADMT mechanisms has been to attempt to minimize the specific manifestations of the environmental effects specifications on the Ada representation of the architecture. Specifically, one may represent the process and assertion portions of the architecture in SADMT in such a way that **no changes** are required in these parts in order to experiment with various execution environments. This requires the following:

(1) that all of the name fields are filled in at initialization so that an external routine **with complete knowledge of the process hierarchy and naming conventions** can uniquely identify a leaf process from its internal identification, and

(2) that all "wait"s that are to be externally controllable are properly identified and parameterized.

Note that transit delays are externally controllable even if just the naming conventions are followed.

5.1. Access to the resource assignment facilities

In this section, we give an overview of how the resource assignment facilities are accessed. What must be done is to define a **resource assignment module** that is called during execution to pick up appropriate values for transit and processing delays. Each such resource assignment module is associated with some platform; the same module could be associated with more than one platform, but at most one module is associated with any particular platform. In fact, the association is by platform designator so a resource assignment module is generally associated with multiple instantiations of a platform process.

Considering Figure 5-1, we can see the ten components of a resource assignment module, six types and four procedures. The "result" of the instantiation of the

```
with PDL_pkg, Resource_Assignment_pkg;
use Resource_Assignment_pkg;
generic
  type process_cache_block is private;
  type process_cache_ptr is access process_cache_block;
  type platform_cache_block is private;
  type platform_cache_ptr is access platform_cache_block;
  type invoke_block is private;
  type invoke_ptr is access invoke_block;
  with procedure initialization_notification(
                        Z: PDL_pkg.PDL_ptr;
                        process_cache: out process_cache_ptr;
                        platform_cache: in out platform_cache_ptr);
  with procedure transit_delay(
                P1, P2: process_cache_ptr;
                cached_EXP: in out RA.PDExp);
  with procedure compute_arrival_time(
                    L_PARAM: invoke_ptr;
                    cached_EXP: in out RA.PDExp;
                    arrival_time: out PDL_pkg.PDL_time_type;
                    this_message_length: integer:= 0);
  with procedure processing_delay_next_state(
                        P: process_cache_ptr;
                        PARAM: RA.P_Delay_parameterization_type;
                        STATE: in out RA.P_Delay_state_type);
package ResourceModuleDefiner_pkg is
  use PDL_pkg;
  procedure define_resource_allocation(P_designator: PDL_string_ptr);
end ResourceModuleDefiner_pkg;
```

Figure 5-1. Specification for the *ResourceModuleDefiner_pkg*.

ResourceModuleDefiner_pkg is a single procedure *define_resource_allocation*. A call to this procedure will associate the resource assignment module formed from the ten components with any platforms designated by the input parameter. Let us very briefly consider what each of the components do.

First note that six types are required to parameterize the generic. Actually, only the access types are needed internally but this form is used to ensure that the types specified are actually access types. Discussion of the *invoke_block* and *invoke_ptr* types will only be meaningful subsequently so we do not discuss them here. What we shall discuss here are the *process_cache_ptr* and *platform_cache_ptr*. Since the procedures specified will be used with an indeterminate number of actual platforms and likewise an indeterminate number of processes on each platform[1], the procedures must be written so that any internal state is associated with the platforms and processes to be manipulated and not declared locally. The driver assists the procedures in doing this by holding certain pointers for the benefit of the procedures. Specifically, the driver holds a pointer of type *platform_cache_ptr* as part of its description of a platform and makes this value available during process initialization. Similarly, the driver holds a value of type *process_cache_ptr* as part of its description of every process. The driver makes this value available to the procedures in order to designate any referrent process. This is mechanism that the procedures must use to maintain local state information.

In addition to the types, there are four procedures that are supplied to form a resource allocation module. The first of the four is the *initialization_notification* procedure. This procedure is called one time for each process during platform/process initializtion. Its purpose is to allow the

[1] Remember that different parameterizations of a platform may have different numbers of processes!

local state blocks to be setup on a per process basis. The second of the four procedures is the transit_delay procedure. This procedure is called once for each internal link during platform/process initialization. Its purpose is to associate a particular distribution with the transit delay of that particular internal link. In many (perhaps most) cases, the distribution is used without further intervention from the resource assignment module. However, the third procedure *compute_arrival_time* is used to model certain situation that are not well handled by the distribution method. The last procedure of the four is the *processing_delay_next_state* procedure. It is called by the driver whenever a process semantics issues the *processing_delay_wait* form of the wait procedure, as explained below. These four procedures together specify all of the resource assignment information associated with the platform.

5.2. A Running Example

In order to illustate the concepts being described throughout this chapter, a running example is presented based on the simple example of Chapter 2. First, we assume that the transit delay for message traffic destined for *RW_proc* is not to be zero. Two different scenarios are explored: first, an exponential random transit delay, and second, a transit delay based on contention. Next, we assume that transit delay for messages destined for the first of the three *simple_proc*s is to be a constant multiple of the message length. Last, we will assume that the processing delay associated with each *simple_proc* is the sum of a constant component and a variable component that is exponentially distributed with the mean given by the parameterization. Further, the *simple_proc*s must compete for a single resource upon which this processing is performed. We further assume that the constant component is high priority and the variable component at lower priority. Also, *simple_proc(2)* has priority over the other two. Next, we turn to the issue of how probability distributions are specified.

```
with PDL_pkg, Resource_Assignment_pkg;
package RA_PDExp_Examples is
  use PDL_pkg;
  function PDL_duration(f: float)
     return PDL_duration_type renames timing_ops.PDL_Duration;
  use Resource_Assignment_pkg.RA;
  use PDExp_routines;
  a, b, c, d, d2, e, e2: PDExp; --just a bunch of PDS
  w1, w2: PDL_duration_type; --
end RA_PDExp_Examples;
package body RA_PDExp_Examples is
begin
  a:= CONST(5.0); --a constant duration of 5 seconds
  b:= UNIFORM(4.0, 6.0); --a uniform random duration
  c:= CONST(4.0) + CONST(2.0) * UNIFORM(0.0, 1.0, SEED => 233); --the same distribution
  d:= b + c; --distributions can be combined
  e:= EXP(5.0) + CONST(3.0); --wait in line; then constant processing
  e2:= EXP(5.0) + CONST(3.0) * MESSAGE_LENGTH; --wait; then 3.0/byte
  set_seed(b, 233); --set the seed explicitly
  randomize(a); --randomizes the seeds in the leafs of a
  randomize(b); --randomizes for b, this also affects d;
  d2:= copy(d); --no interference between d and d2;
  w1:= PDL_duration(select_from(a)); --draw from the distrb. spec'd by a
  w2:= PDL_duration(select_from(a)); --draw again from a's distribution
end RA_PDExp_Examples;
```

Figure 5-2. Examples of *PDExps*.

5.3. Specifying Distributions for Transit and Processing Delays

The need to specify a distribution from which to draw durations for transit and processing delays is common in simulations. SADMT provides a facility for defining expressions called **probabilistic duration expressions** (*PDExps*) whose primitives are random variates from frequently encountered distributions. Each evaluation of a *PDExp* causes a new set of variates to be drawn and combined according to the operators of the expression. Thus, the *PDExp* most likely supplies a different value each time it is evaluated. Importantly, each primitive contains its own seed for the internal pseudorandom number generator so that the author can control the reproduceability of the simulation. Control procedures are provided for fine-grained control over the seeds.

An example demonstrating the definition of several *PDExps* is shown in Figure 5-2; refer to Appendix G for the complete Ada specification of the operators used. Several examples are shown here of defining various distributions. Note that the distribution expressions can be combined using the normal arithmetic operations. There are two examples of the randomize procedure. The point is that, since *d* is built up from named subcomponents, "randomization" can be applied to exactly the components desired or *en masse*. The "copy" function allows two independent generators to be defined. If the program instead specified "d2:= d;", then performing a *select_from(d2)* will affect the sequence generated by calls to *select_from(d)*.

5.4. Identifying the processes and internal links

In order for the procedures of a resource assignment module to recognize the various platforms and internal links of a platform, the *initialization_notification* procedure must use the information provided in the *set_process_parent* call of a processes initialization to uniquely identify the processes; for example, the subprograms defined in the *process_data_extractors* package of the *Resource_Assignment_pkg* package (see can be used to determine such things as the process' level and process id as well as the string-valued identification. In addition, this procedure must allocate space for process-local and platform-local state. The parameters to the *initialization_notification* procedure are (1) the *PDL_ptr* of the process being initialized, (2) an *out* parameter that the procedure will initialize to contain a *process_cache_ptr*, and (3) an *inout* parameter that is the *platform_cache_ptr* associated with the platform. On the call to the first process within any platform, the *platform_cache* variable will have been initialized to *null*. Importantly, the driver supplies only the *process_cache_ptr* on subsequent calls to identify a process; thus, the *initialization_notification* procedure should take care to cache away both the *PDL_ptr* and the *platform_cache_ptr* for subsequent use.

```
type leaflink_types is (notofinterest, rwproc, simpleproc1);
type platform_cache_block is record
  a_machine: resource;
  resource_list: List_Of_Resources;
end record;
type platform_cache_ptr is access platform_cache_block;
type ArrayOf_List_Of_Priorities is array(integer range <>) of List_Of_Priorities;
type process_cache_block is record
  PDL: PDL_ptr;
  PLATFORM: platform_cache_ptr;
  link_type_indicator: leaflink_types:= notofinterest;
  priority_lists: ArrayOf_List_Of_Priorities(1 .. 2);
end record;
type process_cache_ptr is access process_cache_block;
procedure initialization_notification(
                    Z: PDL_pkg.PDL_ptr;
                    process_cache: out process_cache_ptr;
                    platform_cache: in out platform_cache_ptr) is
  use Resource_Assignment_pkg.process_data_extractors;
begin
  if Node_Type(Z) /= leaf then
    return;
  end if;
  if platform_cache = null then
    platform_cache:= new platform_cache_block;
    Start_a_Resource_List(platform_cache.resource_list,
     platform_cache.a_machine);
  end if;
  process_cache:= new process_cache_block;
  process_cache.PDL:= Z;
  process_cache.PLATFORM:= platform_cache;
  if Name(Z).all = "RW_proc" then
    process_cache.link_type_indicator:= rwproc;
  else
    if Discr_Name(Z).all = "1" then
      process_cache.link_type_indicator:= simpleproc1;
    end if;
    if Discr_Name(Z).all = "2" then
      Start_a_Priority_List(process_cache.priority_lists(1), 10, 9);
      Start_a_Priority_List(process_cache.priority_lists(2), 6, 5);
    else
      Start_a_Priority_List(process_cache.priority_lists(1), 9, 9);
      Start_a_Priority_List(process_cache.priority_lists(2), 5, 5);
    end if;
  end if;
  return;
end initialization_notification;
```

Figure 5-3. Example *initialzation_notification* Procedure and Necessary Types.

Let us turn for the moment to an appropriate *initialization_notification* procedure for our running example (see Figure 5-3). Recall that we are wanting to change the transit delay on messages targeted for *RW_proc* and each of the *simple_proc(1)*'s. (The initialization procedure and its **full** context is shown in Appendix J.) First note that the procedure allocates and caches the appropriate values as described above. We defer to a discussion until later of the *Resource* and *Priority* lists and the resources. The remaining field that is initialized is the *link_type_indicator* of the *process_cache_block*. This procedure sets this field to one of three values depending on whether the process is (1) the *rw_proc*, (2) a *simple_proc(1)*, or (3) neither of these. Having set this value in the *process_cache_block*, subsequent invocations of procedures in the resource

allocation module will be able to identify these processes without again refering to their external names. How this is used is the subject of the next section.

5.5. Modeling Transit Delays

When an internal link is first established, the driver invokes a the *transit_delay* procedure to associate a *PDExp* with that link. If the *transit_delay* procedure returns a *null PDExp*, then the system default of zero transit delay is used. Otherwise, the *PDexp* returned will be evaluated (*i.e.*, "selected from") to obtain the transit delay whenever a message traverses the internal link. Notice that the *transit_delay* procedure is not invoked each time a message traverses the link; only when the link is established. Thus, special mechanisms must be used to cause the transit delay to adapt to the load as explained below.

A simple example

A simple example of a *transit_delay* procedure is shown in Figure 5-4. It assumes that the previous *initialization_notification* procedure was used. The procedure uses the cached values (*link_type_indicator*) computed by the initialization procedure to distiguish quickly whether a *PDExp* is to be associated with the link. The procedure specifies that the transit delay on the link into *rw_proc* is a constant 10.0 seconds and that the transit delay on links into the *simple_proc(1)*s are to be drawn from an exponential distribution with a mean of 3.0 seconds. Note that the procedures specify three independent exponential random number generators--one each time a link is established into a *simple_proc(1)*.

Adjusting for Load

Since the *transit_delay* function is invoked only when the link is established, it is considerably more difficult to specify transit delays that are adjusted for load than those that are load independent. There are three ways to compensate for this. The most powerful is to change the model of the system so that the network is modeled as a process. The least powerful (but most often acceptable) technique is to use *PDExp* functionals, specifically the "MESSAGE_LENGTH" form of a *PDExp*. The last technique supported by SADMT allows the modeler to supply a procedure to compute the arrival time directly. Suppose that the transit delay to be modeled in the the *RW_proc* has the property that it takes five seconds unless the messages are too closely spaced. In this case, the messages queue up. To model this situation, the modeler uses the INVOKE form of a *PDExp*. Whenever, the PDExp evaluator encounters an INVOKE, it calls the *compute_arrival_time* procedure using the same argument as was supplied by INVOKE. Since different siutations will call for different state information, the INVOKE

```
procedure transit_delay(P1, P2: process_cache_ptr; cached_EXP: in out PDExp) is
  use PDExp_routines;
begin
  case P2.link_type_indicator is
    when rwproc =>
      cached_Exp:= CONST(10.0);
    when simpleproc1 =>
      cached_Exp:= EXP(3.0);
    when notofinterest =>
      cached_Exp:= null;
  end case;
  return;
end transit_delay;
```

Figure 5-4. Example of a Simple *transit_delay* Procedure.

form of a *PDExp* must be obtained by instantiating the *InvokeDefiner* package with the appropriate types as shown in Figure 5-5. This would be modeled using the INVOKE function as the distribution and providing an appropriate function for computing the arrival time directly. The only state required in this case is the next possible arrival time for a message; thus, the *invoke_block* type declares space to hold this value. In Figure 5-6 and Figure 5-7, the code is depicted for modeling the case.

5.6. Processing Delays

In the previous section, transit delay modeling is described. While it is possible to effect relatively fine-grained delay behaviour by using transit delays, there are limits to what can be modeled. In particular, delays can only be modeled using transit delays when the arrival time of the message can be computed based only on the system state at message emission time. In situations where routing and delivery are affected by downstream resource contention, transit delays may not be used; instead, the message delivery system must be modeled explicitly as a SADMT process.

SADMT processes can have arbitrarily complex delay behaviour based on the various "wait" primitives already described in Chapter 2. However, if these previously described primitives are used, then it is difficult for the processing behaviour to be controlled externally. In this case, "difficult" means that a special Ada program must be written and compiled that has the appropriate delay behaviour and this program becomes part of the architectural specification. In many cases, this effectively results in the structure of the underlying environment (the run-time system, operating system, and hardware) becoming completely intermixed with the system's structural specification; clearly, it is advantageous to keep these specifications separate. This section describes the facilities offered by SADMT for connecting a parametric specification in a

```
with Resource_Assignment_pkg;
use Resource_Assignment_pkg;
generic
  type invoke_block is private;
  type invoke_ptr is access invoke_block;
package InvokeDefiner is
  function INVOKE(IBP: invoke_ptr) return RA.PDExp;
end InvokeDefiner;
```

Figure 5-5. *InvokeDefiner* Generic.

```
package alternative is
  type invoke_block is record
    case_indicator: integer;
    earliest_arrival_time. PDL_time_type;
  end record;
  type invoke_ptr is access invoke_block,
  procedure transit_delay_2(
                P1, P2: process_cache_ptr;
                cached_EXP: in out PDExp);
  procedure compute_arrival_time(
                I_PARAM: invoke_ptr;
                cached_EXP: in out PDExp;
                arrival_time: out PDL_pkg.PDL_time_type;
                this_message_length: integer:= 0);
end alternative;
```

Figure 5-6. Setup for an Alternative *transit_delay* Procedure.

```
separate(RA_example.alternative)
procedure transit_delay_2(P1, P2: process_cache_ptr; cached_EXP: in out PDExp) is
  package my_INVOKE is new InvokeDefiner(invoke_block, invoke_ptr);
  use my_INVOKE;
  use PDExp_routines;
begin
  case P2.link_type_indicator is
    when rwproc =>
      cached_Exp:= INVOKE(new invoke_block'(53, 0));
    when simpleproc1 =>
      cached_Exp:= EXP(3.0);
    when notofinterest =>
      cached_Exp:= null;
  end case;
  return;
end transit_delay_2;
separate(RA_example.alternative)
procedure compute_arrival_time(
                  I_PARAM: invoke_ptr;
                  cached_EXP: in out PDExp;
                  arrival_time: out PDL_pkg.PDL_time_type;
                  this_message_length: integer:= 0) is
  FIVE_SECONDS: constant PDL_duration_type:= PDL_duration_type(5 *
   PDL_ticks_per_second);
  earliest_arrival_time: PDL_time_type
   renames I_PARAM.earliest_arrival_time;
  t: PDL_time_type;
  use timing_ops;
begin
  case I_PARAM.case_indicator is
    when 53 =>
      t:= Current_PDL_time + FIVE_SECONDS;
      if earliest_arrival_time > t then
        t:= earliest_arrival_time;
      end if;
      earliest_arrival_time:= t + FIVE_SECONDS;
      arrival_time:= t;
    when others =>
      null;
  end case;
  return;
end compute_arrival_time;
```

Figure 5-7. Alternative *transit_delay* procedure and corresponding *compute_arrival_time* procedure.

process' semantics to a separately (conceived and) constructed procedure in the resource assignment module. Note that while it is possible to write a procedure based on what has already been presented, using the facilities described here will likely result in a cleaner interface and reduced difficulty in creating the appropriate code.

5.6.1. Specifying Parameterized Processing Delays

The first step in creating a parameterized semantic model is to place into the model code parameterized versions of the wait primitive instead of the unparameterized form presented earlier. The specification for the parameterized form of the wait procedure is the following.

```
procedure processing_delay_wait(
                Op: PDL_string_ptr;
                NumericParams: RA.ArrayOf_Float;
                StringParams: RA.ArrayOf_PDLstringptr;
                Default_Delay: PDL_timing.PDL_duration_type;
                Time_out: PDL_timing.PDL_duration_type;
                P: PDL_ptr);
```

Please refer to the description of the *PDL_pkg* package found in the appendices for contextual details on the types. What is provided by the arguments to the wait procedure is a "parameterized operation." Exactly what the parameterization is depends on the operation being parameterized. For example, one might desire to specify that a process incurs a processing delay that is equal to the amount of time required to perform a 1024-point complex FFT. Or, one may need to specify a delay that depends on the amount of time required to join two databases that are geographically distributed. Specifications for these cases are shown in Figure 5-8.

Considering this, one should first note that while one can obviously make these specifications highly mnemonic, the SADMT driver does not "understand" them. Rather, the intelligence required for understanding how these parameterized operations are translated into processing delays is supplied by the *processing_delay_next_state* procedure. If no such module is provided or if the module declines to provide a translation, then the default delay is used.

A second important issue is the type of the parameters. Specifically, one may question the use of "pointers to strings" instead of using the strings directly. The reason for this is related to another issue: where are the string constant definitions actually placed in the SADMT representation of the architecture. The answer to the second question is that they should ordinarily be placed in a separate package that is "with-and-use"-ed by both the process' task body and by the *processing_delay_next_state* procedure of the platform. If this is done, a considerable space savings is obtained for replicated processes. In addition, a considerable time savings is possible since a resource assignment module does not have to actually read the characters of the string in order to know its contents. Higher level routines that use string parameters directly can be written by users that desire them.

```
with PDL_pkg;
package Simple_PDWait_Example is
end Simple_PDWait_Example;
package body Simple_PDWait_Example is
  use PDL_pkg;
  Z: PDL_ptr;
  package my_waits is new Wait_pkg(Z);
  use my_waits;
  C_FFT: constant PDL_string_ptr:= new string'("C-FFT");
  JOIN: constant PDL_string_ptr:= new string'("JOIN");
  ASSETS_DB: constant PDL_string_ptr:= new string'("ASSETS_DB");
  MAINT_DB: constant PDL_string_ptr:= new string'("MAINT_DB");
begin
    .
    .
  wait(C_FFT, NumericParams => (1 => 1024.0));
    .
    .
  wait(JOIN, StringParams => (ASSETS_DB, MAINT_DB));
    .
end Simple_PDWait_Example;
```

Figure 5-8. Examples of Calls to the *processing_delay_wait* Procedure.

5.6.2. The SADMT View of Processing Delays

Unsurprisingly, SADMT views the operations for which processing delay is to be modeled as transition graphs. At any node of the graph, the calling process must acquire certain resources, called the **resource set,** and must delay for a period of time called the **holding period.** The resources are released at the end of the holding period. If the process is unable to acquire its resource set then it suspends until it can. Having delayed for the holding period and having released its resource set, the process will either enter into a new state or complete the processing delay. Of course, the details are important. First, the total amount of time spent in a processing delay may never exceed the timeout value specified. Second, the resource set for any particular state may be empty. Third, any process acquires and holds a resource with a specific priority; the priority associated with each resource in the resource set could be different. If a processor loses control of a resource to a higher priority process, it forfeits all of its resources. Actually, the acquisition and holding priority of a process in some state with respect to a given resource may also be different. This facilitates the simulation of various scheduling disciplines by the resource allocation module.

The SADMT driver does not actually implement the transition graph; rather, the driver takes responsibility only for (1) reporting to the *processing_delay_next_state* procedure that a new state is to be entered and why a state's holding period was cut short, and (2) for actually holding the resources on behalf of the *processing_delay_next_state* procedure. Note that a resource assignment module may declare resources and make lists of them but may not in fact seize them. Only the driver can seize resources; thus, orderly control over scheduling priorities and the like is assured.

The driver, in fact, implements one additional consideration– piping the arguments from the *processing_delay_wait* (or just "wait") call into the *processing_delay_next_state* procedure of the resource assignment module. Communication into the *processing_delay_next_state* procedure is in three parts:

(1) the cache_ptr of the process making the call.

(2) a block describing the arguments.

(3) a block for communicating state information.

Of the three, the first two are one way (*i.e.*, *in*-parameters) and the last is two-way (*i.e.*, *in out*) since the state information must include not only status information to be acted upon by the *processing_delay_next_state* procedure but also information so that the *processing_delay_wait* procedure knows when to return.

In order to understand how to code a *processing_delay_next_state* procedure, it is necessary to understand more completely the function of the *processing_delay_wait* procedure; a surrogate for this procedure is shown in Figure 5-9. The corresponding body is given in Figure 5-10. Notice that the surrogate is represented as a generic; in fact, this similar code is implemented in the *ResourceModuleDefiner_pkg*; the code is relatively straightforward. First, the routine obtains

```
with PDL_n_Port_pkg;
use PDL_n_Port_pkg;
package support_PDWAIT_surrogate is
  procedure all_or_nothing_seize(
                    STATE: in out RA.P_Delay_state_type;
                    Time_out: PDL_timing.PDL_duration_type);
end support_PDWAIT_surrogate;
```

Figure 5-9. Specification for the *PDWait* Surrogate

```
package body PDWAIT_surrogate is
  procedure processing_delay_wait(
                    Op: PDL_string_ptr;
                    NumericParams: RA.ArrayOf_Float;
                    StringParams: RA.ArrayOf_PDLstringptr;
                    Default_Delay: PDL_timing.PDL_duration_type;
                    Time_out: PDL_timing.PDL_duration_type;
                    P: PDL_ptr) is separate;
end PDWAIT_surrogate;
package body support_PDWAIT_surrogate is
  procedure all_or_nothing_seize(
                    STATE: in out RA.P_Delay_state_type;
                    Time_out: PDL_timing.PDL_duration_type) is
    use PDL_timing;
    use timing_ops;
    use DSS;
    entry_time, exit_time, good_exit_time: PDL_time_type;
    interval: PDL_duration_type;
  begin
    entry_time:= Current_PDL_time;
    if we can seize all the resources at the priorites indicated then
      if STATE.holding_period < Time_out then
        interval:= STATE.holding_period;
      else
        interval:= Time_out;
      end if;
      good_exit_time:= entry_time + interval;
      hold each resource with the priority indicated
            for the interval computed;
      exit_time:= Current_PDL_time;
      STATE.actual_period:= exit_time - entry_time;
      if exit_time < good_exit_time then
        STATE.status:= RA.preempted;
      elsif interval = STATE.holding_period then
        STATE.status:= RA.normal;
      else
        STATE.status:= RA.timed_out;
      end if;
    else
      suspend waiting either for a change in resource
          ownership or until the time-out expires;
      exit_time:= Current_PDL_time;
      STATE.actual_period:= 0;
      if (exit_time - entry_time) >= Time_out then
        STATE.status:= RA.timed_out;
      else
        STATE.status:= RA.preempted;
      end if;
    end if;
  end all_or_nothing_seize;
end support_PDWAIT_surrogate;
```

Figure 5-10. Body for the *PDWait* Surrogate.

and types the *cache_ptr* for the calling process. Next, a *parameterization_block* (*i.e.*, one that contains the parameters) is constructed. This type is defined in the internal portion of the *Resource_Assignment_pkg* (see For convenience, its specification is the following.

```
type P_Delay_parameterization_type(NumNumericParams:
 integer; NumStringParams: integer) is record
  Op: PDL_string_ptr;
  NumericParams: ArrayOf_Float(1 .. NumNumericParams);
  StringParams: ArrayOf_PDLstringptr(1 .. NumStringParams);
  Time_out_time: PDL_time_type;
end record;
```

Third, a *state_block* is constructed. Unlike the *parameterization_block*, the function of various fields of the *state_block* are nonobvious; so we will describe them in some detail. The specification of the *state_block* is the following.

```
type P_Delay_status_type is (initial, normal, timed_out, preempted,
 do_return, defer);
type resource_list_block;
type resource_list_ptr is access resource_list_block;
subtype List_of_Resources is resource_list_ptr;
type priority_list_block;
type priority_list_ptr is access priority_list_block;
subtype List_of_Priorities is priority_list_ptr;
type P_Delay_state_type is record
  state: integer;
  resources: List_of_Resources;
  priorities: List_of_Priorities;
  holding_period: PDL_duration_type;
  actual_period: PDL_duration_type;
  status: P_Delay_status_type;
end record;
```

The component crucial to understanding the rest is the "status" component. Initially, this component is set to the "initial" value. The *processing_delay_next_state* procedure can signal the end of the state machine by changing the status component to *do_return*. Alternatively, the *processing_delay_next_state* procedure can signal that the default delay should be used by setting the status variable to *defer*. In "normal" operation, status is *normal* until a state terminates early either because the *Time_out* has been reached or because the process was unable to obtain and hold the resources throughout the holding period. In the former case, the status is set to *timed_out*; in the latter case, the value is *preempted*. A better understanding of how the *all_or_nothing_seize* procedure operates with the status variable can be reached by studying the pseudocode given for this procedure in Figure 5-11. The *state* component of the state_block is an integer that the *processing_delay_next_state* procedure may use for any purpose whatever. Normally, it would be used to hold state information; however, the state information could also be held in the *process_cache*. The next four parameters-- *resources*, *priorities*, *holding_period*, and *actual_period*-- are used to interface with the *all_or_nothing_seize* procedure. Specifically, *resources* and *priorities* are lists of resources and priorities that the driver will try to obtain for the benefit of the *processing_delay_next_state* procedure. *Holding_period* is the holding period; when the *processing_delay_next_state* procedure is invoked and the status is *preempted*, the *actual_period* component conatins the duration that the process actually held the resources.

5.6.3. An Example processing_delay_next_state Procedure

It remains only to present the *processing_delay_next_state* procedure of the example shown in Figure 5-12. The first piece of the code performs the initialization. The interesting point is that the resource list of the *state_block* is setup in the initialization. In this example, there is only one resource involved; in a more complicated case, the resource set could be different for each state. In the case statement that follows, three cases are distinguished. In the *normal* case, the procedure moves to a new state and picks up the correct priority list for the state; these priority lists were setup in the *initialization_notification* procedure. As the procedure advances into a third state, it signals a return. In the *preempted* case, the procedure subtracts the amount of work actually accomplished on this iteration from the total amount. The status is then returned to *normal*. In the *others* case (which includes *timed_out*), the procedure simply signals a return.

```
separate(PDWAIT_surrogate)
procedure processing_delay_wait(
                    Op: PDL_string_ptr;
                    NumericParams: RA.ArrayOf_Float;
                    StringParams: RA.ArrayOf_PDLstringptr;
                    Default_Delay: PDL_timing.PDL_duration_type;
                    Time_out: PDL_timing.PDL_duration_type;
                    P: PDL_ptr) is
  use PDL_timing;
  use timing_ops;
  use DSS;
  use RA;
  function cast_to_cache_ptr is
     new UNCHECKED_CONVERSION(PDL_magic_ptr,process_cache_ptr);
  PCP: process_cache_ptr;
  subtype P_Delay_param_type is RA.P_Delay_parameterization_type;
  subtype P_Delay_state_type is RA.P_Delay_state_type;
  PARAM: P_Delay_param_type(NumericParams'Last, StringParams'Last);
  STATE: P_Delay_state_type;
begin
  PCP:= cast_to_cache_ptr(P.ra_process_cache);
  PARAM:= P_Delay_param_type'(
                    NumericParams'Last,
                    StringParams'Last,
                    Op => Op,
                    NumericParams => NumericParams,
                    StringParams => StringParams,
                    Time_out_time => Current_PDL_time + Time_out);
  STATE:= P_Delay_state_type'(
                    state => 0,
                    resources => null,
                    priorities => null,
                    holding_period => 0,
                    actual_period => 0,
                    status => RA.initial);
--  in the real code there is a check here for
--  the existence of a resource assignment module
  loop
    processing_delay_next_state(PCP, PARAM, STATE);
    if STATE.status > RA.preempted then
      if STATE.status = RA.defer then
        if Time_out < Default_Delay then
          wait(Time_out, P);
        else
          wait(Default_Delay, P);
        end if;
      end if;
      return;
    end if;
    all_or_nothing_seize(STATE, PARAM.Time_out_time - Current_PDL_time);
  end loop;
end processing_delay_wait;
package body support_PDWAIT_surrogate is
  procedure all_or_nothing_seize(
                    STATE: in out RA.P_Delay_state_type;
                    Time_out: PDL_timing.PDL_duration_type) is
    use PDL_timing;
    use timing_ops;
    use DSS;
    entry_time, exit_time, good_exit_time: PDL_time_type;
    interval: PDL_duration_type;
  begin
    entry_time:= Current_PDL_time;
    if we can seize all the resources at the priorites indicated then
```

```
        if STATE.holding_period < Time_out then
          interval:= STATE.holding_period;
        else
          interval:= Time_out;
        end if;
        good_exit_time:= entry_time + interval;
        hold each resource with the priority indicated
                for the interval computed;
        exit_time:= Current_PDL_time;
        STATE.actual_period:= exit_time - entry_time;
        if exit_time < good_exit_time then
          STATE.status:= RA.preempted;
        elsif interval = STATE.holding_period then
          STATE.status:= RA.normal;
        else
          STATE.status:= RA.timed_out;
        end if;
      else
        suspend waiting either for a change in resource
            ownership or until the time-out expires;
        exit_time:= Current_PDL_time;
        STATE.actual_period:= 0;
        if (exit_time - entry_time) >= Time_out then
          STATE.status:= RA.timed_out;
        else
          STATE.status:= RA.preempted;
        end if;
      end if;
    end all_or_nothing_seize;
  end support_PDWAIT_surrogate;
```

Figure 5-11. Surrogate for the *Seize* Routine.

```
procedure processing_delay_next_state(
                    P: process_cache_ptr;
                    PARAM: P_Delay_parameterization_type;
                    STATE: in out P_Delay_state_type) is
  use timing_ops;
begin
  if STATE.status = initial then
    STATE.state:= 0;
    STATE.resources:= P.PLATFORM.resource_list;
    STATE.status:= normal;
  end if;
  case STATE.status is
    when normal =>
      STATE.state:= STATE.state + 1;
      if STATE.state > P.priority_lists'last then
        STATE.status:= do_return;
        return;
      end if;
      STATE.priorities:= P.priority_lists(STATE.state);
      if STATE.state = 1 then
        STATE.holding_period:= PDL_duration(3.0);
      else
        STATE.holding_period:= PDL_duration(PARAM.NumericParams(1));
      end if;
    when preempted =>
      STATE.holding_period:= STATE.holding_period - STATE.actual_period;
      STATE.status:= normal;
    when others =>
      STATE.status:= do_return;
  end case;
end processing_delay_next_state;
```

Figure 5-12. Example *processing_delay_next_state* Procedure.

5.7. Instantiating a Resource Assignment Module

In Section 5.1, the strategy for instantiating resource assignment modules was outlined. All that is left to say is that two examples of instantiating such modules are found in Appendix J. Note that in case of the simpler communications behaviour, no *invoke_ptr* type is needed. The external *Resource_Assignment_pkg* shown in Appendix G contains types and a procedure in its DUMMY_INVOKE package that can be used whenever the INVOKE facilities are not needed.

APPENDIX A

April 1988 Specification for the PortDefiner Package

The *PortDefiner_pkg* defines the port types and the operations upon the port types. This generic package is instantiated with the message type *T* and a procedure to print the contents of a message of type *T*. The *Debug_Port_id* is a string used to identify the port type, and *Max_Work_Space* defines the size of the work space for the port type.

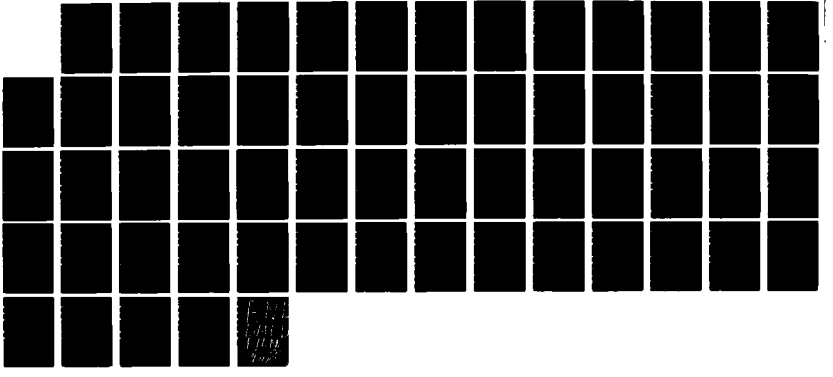
Specification of the PortDefiner_pkg Package

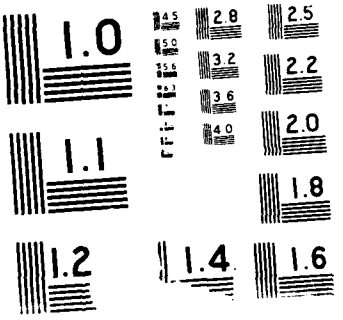
```
with PDL_n_PORT_pkg;
generic
  type T is private;
  with procedure print_port_procedure(
                    Port_Data: T;
                    indent: integer:= 20);
  Port_type_name: string:= "anonymous port type";
  Debug_Port_id: string:= "??PORT??";
  Max_Work_Space: integer:= 200;
package PortDefiner_pkg is

  --PORT TYPES
  type T_port(D: PDL_n_Port_pkg.Port_Direction) is record
    PORT: PDL_n_PORT_pkg.Port_ptr:=
     PDL_n_PORT_pkg.Port_Stuff.new_Port_block(D);
  end record;
  type T_ipptr is access T_port(PDL_n_PORT_pkg.inport);
  type T_opptr is access T_port(PDL_n_PORT_pkg.outport);
  type T_sipptr is access T_port(PDL_n_PORT_pkg.selectable_inport);
  type T_sopptr is access T_port(PDL_n_PORT_pkg.selectable_outport);
  type T_ptr is access T;


  package Procedures is

   --INIT and LINK
    procedure initialize(
              PP: T_ipptr;
              PRC: PDL_n_PORT_pkg.PDL_ptr;
              Characteristics: string:= "";
              port_name: string:= "");
    procedure initialize(
              PP: T_opptr;
              PRC: PDL_n_PORT_pkg.PDL_ptr;
              Characteristics: string:= "";
              port_name: string:= "");
    procedure initialize(
              PP: T_sipptr;
              PRC: PDL_n_PORT_pkg.PDL_ptr;
              Characteristics: string:= "";
              port_name: string:= "");
    procedure initialize(
              PP: T_sopptr;
              PRC: PDL_n_PORT_pkg.PDL_ptr;
              Characteristics: string:= "";
```

```
            port_name: string:= "");
 procedure internal_link(FromPort: T_opptr; ToPort: T_ipptr);
 procedure internal_link(FromPort: T_sopptr; ToPort: T_sipptr);
 procedure inherited_link(ParentPort, ChildPort: T_ipptr);
 procedure inherited_link(ChildPort, ParentPort: T_opptr);
 procedure inherited_link(ParentPort, ChildPort: T_sipptr);
 procedure inherited_link(ChildPort, ParentPort: T_sopptr);

---EMITS, TEST, and CONSUME
--- Note: A PORTLIST IS AN ARRAY OF PORT_PTRs
 procedure emit(ToPort: T_opptr; Data: T);
 procedure emit(ToPort: T_sopptr; Data: T);
 procedure emit(
         ToPort: T_sopptr;
         Data: T;
         DestList: PDL_n_PORT_pkg.PortList);
 function Port_length(FromPort: T_ipptr) return integer;
 function Port_length(FromPort: T_sipptr) return integer;
 function Port_empty(FromPort: T_ipptr) return Boolean;
 function Port_empty(FromPort: T_sipptr) return Boolean;
 function Port_timestamp(
              FromPort: T_ipptr;
              n: integer:= 1)
    return PDL_n_PORT_pkg.PDL_timing.PDL_time_type;
 function Port_timestamp(
              FromPort: T_sipptr;
              n: integer:= 1)
    return PDL_n_PORT_pkg.PDL_timing.PDL_time_type;
 function Port_data(FromPort: T_ipptr; n: integer:= 1) return T;
 function Port_data(FromPort: T_sipptr; n: integer:= 1) return T;
 procedure consume(FromPort: T_ipptr; n: integer:= 1);
 procedure consume(FromPort: T_sipptr; n: integer:= 1);

---DEBUG
 procedure put_pptr(P: T_ipptr; t: string:= "");
 procedure put_pptr(P: T_opptr; t: string:= "");
 procedure put_pptr(P: T_sipptr; t: string:= "");
 procedure put_pptr(P: T_sopptr; t: string:= "");
 procedure Port_Debug(FromPort: T_ipptr);
 procedure Port_Debug(FromPort: T_sipptr);
 procedure Port_Debug(ToPort: T_opptr);
 procedure Port_Debug(ToPort: T_sopptr);
 procedure Clear_Port_Debug(FromPort: T_ipptr);
 procedure Clear_Port_Debug(FromPort: T_sipptr);
 procedure Clear_Port_Debug(ToPort: T_opptr);
 procedure Clear_Port_Debug(ToPort: T_sopptr);
 procedure Enable_Debug_of_Port_type;
 procedure Disable_Debug_of_Port_type;
 procedure Total_Debug_of_Port_type;
 procedure Disable_Total_Debug_of_Port_type;

---Extra Operations
 function "=" (a, b: T_ipptr)
    return Boolean renames PortDefiner_pkg."=";
 function "=" (a, b: T_opptr)
    return Boolean renames PortDefiner_pkg."=";
 function "=" (a, b: T_sipptr)
    return Boolean renames PortDefiner_pkg."=";
 function "=" (a, b: T_sopptr)
    return Boolean renames PortDefiner_pkg."=";

---CONSTRAINTS
 function Base_Type(P: T_port) return T;
 function Ensure_Port(P: T_port) return Boolean;
 function Ensure_Output_Port(P: T_opptr) return Boolean;
 function Ensure_Output_Port(P: T_sopptr) return Boolean;
```

```
    function Ensure_Input_Port(P: T_ipptr) return Boolean;
    function Ensure_Input_Port(P: T_sipptr) return Boolean;
  end Procedures;
  generic
    with function FFF(M: T) return Boolean;
  package PRED is
    task CHECK_task is
      entry check_msg(M: T; Check_Result: out Boolean);
    end CHECK_task;
  end PRED;
end PortDefiner_pkg;
```

AD-A194 356 STRATEGIC DEFENSE INITIATIVE ARCHITECTURE DATAFLOW 2/2
MODELING TECHNIQUE VER (U) INSTITUTE FOR DEFENSE
ANALYSES ALEXANDRIA VA J L LINN ET AL 22 AP 88
UNCLASSIFIED IDA-P-2035 IDA/HQ-87-32623 MDA903-84-C-0031 F/G 15/3 1 NL




1.0
1.1
1.2
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

APPENDIX B

April 1988 Specification for the PDL Package

The *PDL_IO* package defines a group of useful packages and procedures. The package is given below. An example of its use is given in Appendix A.

Specification of the PDL_IO Package

```
package PDL_IO is
  --DEBUG FLAGS
  current_debug_level: integer renames Debug_flags.current_debug_level;
  display_whole_system: Boolean renames Debug_flags.display_whole_system;
  init_debug_level: integer renames Debug_flags.init_debug_level;
  digital_debug_level: integer renames Debug_flags.digital_debug_level;
  analog_debug_level: integer renames Debug_flags.analog_debug_level;
  sched_debug_level: integer renames Debug_flags.sched_debug_level;
  port_debug_level: integer renames Debug_flags.port_debug_level;
  terminate_debug_level: integer renames Debug_flags.terminate_debug_level;
  debug_levels: Debug_flags.debug_array renames Debug_flags.debug_levels;

  --RANDOM
  package random renames PDL_random;

  --IO PACKAGES
  package TXT_IO renames Debug_flags.TXT_io;
  package INT_IO renames Debug_flags.INT_io;
  package TIME_IO is new txt_io.INTEGER_IO(PDL_time_type);
  package DURATION_IO is new txt_io.INTEGER_IO(PDL_duration_type);
  package FLT_IO is new txt_io.FLOAT_IO(float);
  package PROCID_IO is new txt_io.INTEGER_IO(PDL_process_id_type);
  package PD_IO is new txt_io.ENUMERATION_IO(Port_Direction);
  package PNT_IO is new txt_io.ENUMERATION_IO(Process_Node_Type);

  --IO PROCEDURES and FUNCTIONS
  procedure write_process_id(
                Z: PDL_ptr;
                prefix, suffix: string:= "";
                end_of_line: Boolean:= true);
  procedure write_process_full(
                Z: PDL_ptr;
                prefix, suffix: string:= "";
                end_of_line: Boolean:= true);
  function write_it(
            s: string;
            print_value: integer:= 0;
            debug_selector: integer:= 0;
            debug_level: integer:= 0;
            value: integer:= 1)
     return integer;
  function write_it(
            s: string;
            print_value: integer:= 0;
            debug_selector: integer:= 0;
            debug_level: integer:= 0;
            value: Boolean:= false)
     return Boolean;
```

```
  function set_debug_level(
                debug_selector: integer:= 0;
                debug_level: integer:= 0;
                value: integer:= 1)
      return integer;
end PDL_IO;
```

The *PDL_IO* package begins by renaming the debug flags found in the *Debug_flags* package, and continues by defining the *random* package. The *random* package defines a set of random number functions. The specification and brief description of the package is given below.

Specification of the RANDOM Package

```
package RANDOM is

  procedure rand(seed: in out integer);
  --RAND takes as its arguement an integer variable containing the seed,
  --and returns a random integer based on that seed in the seed
  --variable. The integer is in the range 0..(2**16 - 1). This
  --random integer can then be used as the seed in the next call
  --to RAND.

  procedure rand(seed: in out integer; num: out float);
  --This is the same as RAND above, except it also returns a float
  --proportional to the random integer in the range 0.0 .. 1.0.

  procedure normal_rand(seed: in out integer; num: out float);
  ----NORMAL_RAND takes in a integer seed, and returns a uniformly
  ----distributed FLOAT (mean = 0, std. dev. = 1). It also returns
  ----the next value to use as the seed in the seed variable.

  procedure exp_rand(seed: in out integer; num: out float);
  ---EXP_RAND is the same as NORMAL_RAND, but returns a float
  ----with an exponential distribution (mean = 1).

  procedure poisson_rand(
                seed: in out integer;
                num: out integer;
                mean: in float:= 1.0);
  ----POISSON_RAND is similar to exp_rand, but
  ---returns a poisson-distributed integer.
  ---The mean can be changed through the MEAN variable.

end RANDOM;
```

The *PDL_pkg* package makes the useful types and procedures of the *PDL_n_PORT_pkg* package visable to the SADMT processes. The *PDL_n_PORT_pkg* package is **not** given in this document because it is not useful in the specification of SADMT processes. In fact, **a program or package that uses PDL_n_PORT_pkg is not a valid SADMT process, platform, or technology module**. The philosophy is, *"What you don't need to know, you don't get to see."*

Specification of the PDL_pkg Package

```
with PDL_n_PORT_pkg, RegExpOpDefiner_pkg;
package PDL_pkg is
  package PDL_timing renames PDL_n_PORT_pkg.PDL_timing;
  package timing_ops renames PDL_n_PORT_pkg.PDL_timing.timing_ops;
```

```
package PDL_IO renames PDL_n_PORT_pkg.PDL_IO;
subtype PDL_string_ptr is PDL_n_Port_pkg.PDL_string_ptr;
function "=" (l, r: PDL_string_ptr)
   return Boolean renames PDL_n_PORT_pkg."=";
empty_string: PDL_string_ptr:= PDL_n_Port_pkg.empty_PDL_string;
anonymous_name: PDL_string_ptr:= PDL_n_Port_pkg.anonymous_process_name;


--TIME
subtype PDL_time_type is PDL_timing.PDL_time_type;
max_PDL_time: PDL_time_type renames PDL_timing.max_PDL_time;
PDL_ticks_per_second: constant:= PDL_timing.PDL_ticks_per_second;
subtype PDL_duration_type is PDL_timing.PDL_duration_type;
max_PDL_duration: PDL_duration_type renames PDL_timing.max_PDL_duration;
function Current_PDL_Time
   return PDL_time_type renames PDL_n_PORT_pkg.Current_time;

--PORT
subtype Port_ptr is PDL_n_PORT_pkg.Port_ptr;
--A PORTLIST IS AN ARRAY OF PORT_PTRs
subtype PortList is PDL_n_Port_pkg.PortList;

subtype Port_Direction is PDL_n_PORT_pkg.Port_Direction;
function inport return Port_Direction renames PDL_n_PORT_pkg.inport;
function selectable_inport
   return Port_Direction renames PDL_n_PORT_pkg.selectable_inport;
function outport
   return Port_Direction renames PDL_n_PORT_pkg.outport;
function selectable_outport
   return Port_Direction renames PDL_n_PORT_pkg.selectable_outport;

--PORT DEBUG ENABLE/DISABLE
procedure Enable_Port_Debug renames PDL_n_PORT_pkg.PSS.Enable_Port_Debug;
procedure Disable_Port_Debug renames PDL_n_Port_pkg.PSS.Disable_Port_Debug;

--DUMMY DEBUG PRINT ROUTINE
generic
  type T is private;
procedure dummy_print_procedure(Port_Data: T; indent: integer:= 20);

--PROCESS
subtype Process_Node_type is PDL_n_PORT_pkg.Process_Node_type;
function "=" (
        l,
        r: Process_Node_Type)
   return Boolean renames PDL_n_PORT_pkg."=";
function leaf return Process_Node_type renames PDL_n_PORT_pkg.leaf;
function nonleaf
   return Process_Node_type renames PDL_n_PORT_pkg.nonleaf;
function platform
   return Process_Node_type renames PDL_n_PORT_pkg.platform;

subtype PDL_ptr is PDL_n_PORT_pkg.PDL_ptr;
function new_PDL_block(NT: Process_Node_type)
   return PDL_ptr renames PDL_n_PORT_pkg.DSS.new_PDL_block;
procedure set_process_parent(
                Child, Parent: PDL_ptr;
                Process_name: PDL_string_ptr:= anonymous_name;
                Discriminant_name: PDL_string_ptr:= empty_string;
                Process_type_name: PDL_string_ptr:= empty_string;
                Characteristics: PDL_string_ptr:= empty_string) renames PDL_n_PORT_pkg.DSS.set_process_parent;
procedure make_known(P: PDL_ptr) renames PDL_n_PORT_pkg.DSS.make_known;

--FUNCTIONS FOR NAMES
function get_name_of(Z: PDL_ptr)
   return PDL_string_ptr renames PDL_n_PORT_pkg.get_name_of;
```

```
function get_discr_of(Z: PDL_ptr)
   return PDL_string_ptr renames PDL_n_PORT_pkg.get_discr_of;
function get_type_of(Z: PDL_ptr)
   return PDL_string_ptr renames PDL_n_PORT_pkg.get_type_of;
function get_characteristic_of(Z: PDL_ptr)
   return PDL_string_ptr renames PDL_n_PORT_pkg.get_characteristic_of;

---REGULAR EXPRESSIONS
package Event_pkg is
  use PDL_n_PORT_pkg;
  subtype regular_expression is RGEXP.regular_expression;
  function empty_string
     return regular_expression renames RGEXP.empty_string;
  function any_event
     return regular_expression renames RGEXP.any_event;
  function "not" (A: regular_expression)
     return regular_expression renames RGEXP."not";
  function "+" (A, B: regular_expression)
     return regular_expression renames RGEXP."+";
  function "and" (A, B: regular_expression)
     return regular_expression renames RGEXP."and";
  function "or" (A, B: regular_expression)
     return regular_expression renames RGEXP."or";
  function ">" (A, B: regular_expression)
     return regular_expression renames RGEXP.">";
  function closure(A: regular_expression)
     return regular_expression renames RGEXP.closure;
  function any(A: regular_expression)
     return regular_expression renames RGEXP.any;
  function every(A: regular_expression)
     return regular_expression renames RGEXP.every;
  function each(A: regular_expression)
     return regular_expression renames RGEXP.each;

  type RE_sequence is array(integer range <>) of regular_expression;
  package RE_sequence_pkg is
     new RegExpOpDefiner_pkg(integer, RE_sequence);
  function any(AA: RE_sequence)
     return regular_expression renames RE_sequence_pkg.any;
  function every(AA: RE_sequence)
     return regular_expression renames RE_sequence_pkg.every;
  function each(AA: RE_sequence)
     return regular_expression renames RE_sequence_pkg.each;
  function sequence(AA: RE_sequence)
     return regular_expression renames RE_sequence_pkg.sequence;
end Event_pkg;

---WAITS
generic
  Z: PDL_ptr;
package Wait_pkg is
  use PDL_n_PORT_pkg;
  use Resource_Assignment_pkg;
  null_Float_array_1: ArrayOf_Float(1 .. 0);
  null_Float_array: constant ArrayOf_Float(1 .. 0):=
   null_Float_array_1;
  null_PDLstringptr_array_1: ArrayOf_PDLstringptr(1 .. 0);
  null_PDLstringptr_array: constant ArrayOf_PDLstringptr(1 .. 0):=
   null_PDLstringptr_array_1;

  procedure wait_for_initialization(P: PDL_ptr:= Z) renames DSS.wait_for_initialization;
  procedure wait(
          Interval: PDL_duration_type;
          P: PDL_ptr:= Z) renames DSS.wait;
  procedure wait_for_activity(
                  PL: PortList;
```

```
                which_port: out integer;
                StartTime: PDL_time_type:= PDL_n_PORT_pkg.Current_time;
                Time_out: PDL_duration_type:= max_PDL_duration;
                P: PDL_ptr:= Z) renames DSS.wait_for_activity;
    procedure wait_for_activity(
                PL: PortList;
                StartTime: PDL_time_type:= PDL_n_PORT_pkg.Current_time;
                Time_out: PDL_duration_type:= max_PDL_duration;
                P: PDL_ptr:= Z) renames DSS.wait_for_activity_2;
    procedure wait_for_activity(
                StartTime: PDL_time_type:= PDL_n_PORT_pkg.Current_time;
                Time_out: PDL_duration_type:= max_PDL_duration;
                P: PDL_ptr:= Z) renames DSS.wait_for_activity_3;
    procedure wait(
          Op: PDL_string_ptr;
          NumericParams: ArrayOf_Float:= null_Float_array;
          StringParams: ArrayOf_PDLstringptr:= null_PDLstringptr_array;
          Default_Delay: PDL_duration_type:= 0;
          Time_out: PDL_duration_type:= max_PDL_duration;
          P: PDL_ptr:= Z) renames DSS.processing_delay_wait;
  end Wait_pkg;

  --THIS PROCEDURE IS VISABLE HERE SO THAT A CALL TO
  --CAN BE MADE BEFORE THE TASK SEM EXITS. THIS
  --WILL PREVENT AN INVALID TASK FROM HALTING THE
  --SIMULATION. (SADMT PROCESSES ARE NOT TO TERMINATE
  --OUTSIDE OF THE DRIVERS CONTROL. KILL_PROCESS(Z.PDL)
  --WILL CAUSE THE DRIVER TO TERMINATE A PROCESS.
  --KILL_PROCESS IS THE SAME AS WAIT(-1).
  procedure kill_process(P: PDL_ptr);

--CONSTRAINT
  function Ensure_Process(P: PDL_ptr)
     return Boolean renames PDL_n_PORT_pkg.PSS.Ensure_Process;

end PDL_pkg;
```

APPENDIX C

April 1988 Specification for the SADMT Timing Operations

The *PDL_timing* package defines the time and durations types and the constants *max_PDL_time*, *max_PDL_duration*, and *PDL_ticks_per_second*. The package *timing_ops* defines the operations available with the time and durations types.

Specification of the PDL_timing package

```
package PDL_timing is
  package coverup_ops is
    type PDL_time_type is new integer;
    type PDL_duration_type is new integer;
  end coverup_ops;
  subtype PDL_time_type is coverup_ops.PDL_time_type;
  max_PDL_time: constant PDL_time_type:= PDL_time_type'last;
  subtype PDL_duration_type is coverup_ops.PDL_duration_type;
  max_PDL_duration: constant PDL_duration_type:= PDL_duration_type'last;
  PDL_ticks_per_second: constant:= 10000;
  package timing_ops is
    function "=" (l, r: PDL_duration_type)
       return Boolean renames coverup_ops."=";
    function "<" (l, r: PDL_duration_type)
       return Boolean renames coverup_ops."<";
    function ">" (l, r: PDL_duration_type)
       return Boolean renames coverup_ops.">";
    function "<=" (l, r: PDL_duration_type)
       return Boolean renames coverup_ops."<=";
    function ">=" (l, r: PDL_duration_type)
       return Boolean renames coverup_ops.">=";
    function "+" (r: PDL_duration_type)
       return PDL_duration_type renames coverup_ops."+";
    function "-" (r: PDL_duration_type)
       return PDL_duration_type renames coverup_ops."-";
    function "abs" (r: PDL_duration_type)
       return PDL_duration_type renames coverup_ops."abs";
    function "+" (l, r: PDL_duration_type)
       return PDL_duration_type renames coverup_ops."+";
    function "-" (l, r: PDL_duration_type)
       return PDL_duration_type renames coverup_ops."-";
    function "*" (l, r: PDL_duration_type)
       return PDL_duration_type renames coverup_ops."*";
    function "/" (l, r: PDL_duration_type)
       return PDL_duration_type renames coverup_ops."/";
    function "rem" (l, r: PDL_duration_type)
       return PDL_duration_type renames coverup_ops."rem";
    function "mod" (l, r: PDL_duration_type)
       return PDL_duration_type renames coverup_ops."mod";
    function "**" (
          l: PDL_duration_type;
          r: integer)
       return PDL_duration_type renames coverup_ops."**";
    function "=" (l, r: PDL_time_type)
       return Boolean renames coverup_ops."=";
    function "<" (l, r: PDL_time_type)
       return Boolean renames coverup_ops."<";
```

```
  function ">" (l, r: PDL_time_type)
     return Boolean renames coverup_ops.">";
  function "<=" (l, r: PDL_time_type)
     return Boolean renames coverup_ops."<=";
  function ">=" (l, r: PDL_time_type)
     return Boolean renames coverup_ops.">=";
  function "+" (
          left: PDL_time_type;
          right: PDL_duration_type)
     return PDL_time_type;
  function "+" (
          left: PDL_duration_type;
          right: PDL_time_type)
     return PDL_time_type;
  function "-" (
          left: PDL_time_type;
          right: PDL_duration_type)
     return PDL_time_type;
  function "-" (
          left: PDL_time_type;
          right: PDL_time_type)
     return PDL_duration_type;
  function PDL_duration(float_dur: float)
     return PDL_duration_type;
end timing_ops;
```

APPENDIX D

April 1988 Specification for the Cones_n_Platforms package

The *Cones_n_Platforms* package defines the facilities provided to the authors of technology modules. This include the *cone_type, eqn_motion_type, eqn_motion_pkg, physical_stuff_block,* and *Environ_msg_pkg*. Also included is the *interface_pkg* package and the generic *Procedures* package (renames many of the routines of the *interface_pkg*). The *interface_pkg* contains the *PIG_pkg, ConeDefiner_pkg, PlatformDefiner_pkg*, the creator packages, and several useful routines. The *PDL_n_PORT_pkg* package is *with*ed and *use*d by *Cones_n_Platforms* package because the internal workings of the simulation driver are used in the construction of the *Cones_n_Platforms* package. The *PDL_n_PORT_pkg* package is **not** given in this document because it is not useful in the specification of SADMT processes. The useful parts of the *PDL_n_PORT_pkg* package have be renamed and made visable in the *PDL_pkg* package. **An Ada compilation unit that uses PDL_n_PORT_pkg package is not a valid SADMT process, platform, or technology module.**

Specification of the Cones_n_Platforms Package

```
with UNCHECKED_CONVERSION;
with PDL_n_PORT_pkg, PDL_pkg, PortDefiner_pkg, Vector_pkg;
use PDL_n_PORT_pkg, Vector_pkg;
package Cones_n_Platforms is
  --CONVERSION FUNCTIONS
  subtype PDL_magic_ptr is PDL_n_PORT_pkg.PDL_magic_ptr;
  generic
    type T is private;
    type T_ptr is access T;
  package Casting_Functions is
    --here we define the coersion functions so that technology modules
    --can use the untyped communications structure of the analog
    --beaming facility!!
    function CAST_T_ptr_INTO_magic_ptr is
        new UNCHECKED_CONVERSION(
                    T_ptr,
                    PDL_n_PORT_pkg.PDL_magic_ptr);
    function CAST_magic_ptr_INTO_T_ptr is
        new UNCHECKED_CONVERSION(
                    PDL_n_PORT_pkg.PDL_magic_ptr,
                    T_ptr);
    -- this form of casting is safe for a uniprocessor but TMs should be very
    -- careful not to start overwriting data before it is read by the receiver.
  end Casting_Functions;

  --RENAME TRICK TO MAKE PDL_pkg VISIBLE
  package PDL_pkg_zzz renames PDL_pkg;
  package PDL_pkg renames PDL_pkg_zzz;

  use PDL_timing;
  use timing_ops;
  use Message_pkg, Digital_simulation_stuff;
  use PDL_IO;
  use TXT_IO, INT_IO, TIME_IO, PROCID_IO, FLT_IO;
```

```
--CONE TYPE
type cone_type is record
  source_point: point_type;
  indicator_point: point_type;
  half_angle: float;
  blackout_radius: float:= 0.0;
end record;

--HANDY CONSTANTS
origin: constant point_type:= (0.0, 0.0, 0.0);
cone_everything: constant cone_type:= cone_type'(
                              origin,
                              origin,
                              360.0,
                              0.0);
PDL_lunits_per_kilometer: constant:= 1.0 / 12960.0;

--EQUATION OF MOTION AND PHYSICAL STUFF
type eqn_motion_rec;
type eqn_motion_type is access eqn_motion_rec;
type eqn_motion_rec is record
  position: point_type;
  delta_t: PDL_duration_type;
  back_ptr_flag: boolean:= false;
  next_rec: eqn_motion_type:= null;
end record;

package eqn_motion_pkg is
  function new_eqn_motion_rec return eqn_motion_type;
  --this function provides a new record for building
  --representations of equations of motion.
  procedure free_eqn_motion_rec(e: in out eqn_motion_type);
  --this procedure frees a SINGLE eqn_motion_rec, placing it
  --in the global store. If this record points to more used
  --space, that space will NOT be freed, but will be lost.
  procedure free_eqn_motion(e: in out eqn_motion_type);
  --this procedure frees an entire linked list of eqn_motion_rec's.
  --if the list is circularly linked, it transforms it into a
  --flat list in order to free it all at once.
end eqn_motion_pkg;
use eqn_motion_pkg;

type physical_stuff_block is record
  mass: float:= 0.0;
  cross_sect_rad: float:= 1.0;
  eqn_motion: eqn_motion_type:= null;
  current_eqn_segment: eqn_motion_type:= null;
  when_arrived_this_segment: float:= 0.0;
  when_leaving_this_segment: float:= 10000000.0;
  delta_t_this_segment: float:= 10000000.0;
  where_at_start_of_segment: point_type:= origin;
  where_at_end_of_segment: point_type:= origin;
  speed_this_segment: float:= 0.0;
  distance_this_segment: float:= 0.0;
end record;

--DESIGNATOR TYPES
type platform_designator_id_type is range 0 .. 100;
type platform_designator_type is access string;
type cone_designator_type is access string;
subtype dyn_designator_id_type is PDL_n_PORT_pkg.dynamic_designator_id_type;

--ENVIRONMENT MESSAGES AND AND THEIR PORT TYPES (PIG)
package Environ_Msg_pkg is
  --COLLISION MESSAGES
  type Platform_Msg is record
```

```
    designator: platform_designator_type;
    PHYSICAL: physical_stuff_block;
      --physical quantities associated with
      --the colliding platform
    distance: float;
  end record;
  Platform_Msg_Debug_Class: string(1 .. 9):= "$Platform";
  procedure Put_Platform_Msg(P: Platform_Msg; indent: integer:= 20);

  --BEAMING MESSAGES
  type Cone_RetAddr_type is new integer;

  type Cone_Msg is record
    designator: cone_designator_type;
    initiator_id: Cone_RetAddr_type;
    cone_characteristics: cone_type;
    data: PDL_magic_ptr;
  end record;
  Cone_Msg_Debug_Class: string(1 .. 8):= "$Beaming";
  procedure Put_Cone_Msg(C: Cone_Msg; indent: integer:= 20);

  --EVENT MESSAGES
  type Event_Msg is (end_of_eqn_motion, end_of_lifetime);
  Event_Msg_Debug_Class: string(1 .. 6):= "$Event";
  procedure Put_Event_Msg(E: Event_Msg; indent: integer:= 20);

  --PORT PACKAGES FOR EACH OF THE MESSAGE TYPES
  package PDplat is
      new PortDefiner_pkg(
                Platform_Msg,
                Put_Platform_Msg,
                "PLATFORM_MSG",
                Platform_Msg_Debug_Class);
  package PDcone is
      new PortDefiner_pkg(
                Cone_Msg,
                Put_Cone_Msg,
                "CONE_MSG",
                Cone_Msg_Debug_Class);
  package PDevent is
      new PortDefiner_pkg(
                Event_Msg,
                Put_Event_Msg,
                "EVENT_MSG",
                Event_Msg_Debug_Class);

  --PORT TYPES (selectable types are not used by the PIG)
  subtype Platform_Msg_port is PDplat.T_port;
  subtype Platform_Msg_ipptr is PDplat.T_ipptr;
  subtype Platform_Msg_sipptr is PDplat.T_sipptr;
  subtype Platform_Msg_opptr is PDplat.T_opptr;
  subtype Platform_Msg_sopptr is PDplat.T_sopptr;
  subtype Cone_Msg_port is PDcone.T_port;
  subtype Cone_Msg_ipptr is PDcone.T_ipptr;
  subtype Cone_Msg_sipptr is PDcone.T_sipptr;
  subtype Cone_Msg_opptr is PDcone.T_opptr;
  subtype Cone_Msg_sopptr is PDcone.T_sopptr;
  subtype Event_Msg_port is PDevent.T_port;
  subtype Event_Msg_ipptr is PDevent.T_ipptr;
  subtype Event_Msg_sipptr is PDevent.T_sipptr;
  subtype Event_Msg_opptr is PDevent.T_opptr;
  subtype Event_Msg_sopptr is PDevent.T_sopptr;
end Environ_Msg_pkg;
use Environ_Msg_pkg;
use PDplat.Procedures, PDcone.Procedures, PDevent.Procedures;
```

```
--PROCEDURES USED TO INTERFACE WITH THE SIMULATION SYSTEM
--INCLUDING THE PIG_pkg
package interface_procs is
  --PIG
  package PIG_pkg is
    type PIG_block;
    type PIG_type is access PIG_block;

    type PIG_block is record
      PDL: PDL_ptr:= new_PDL_block(pig);
      collisions: Platform_Msg_opptr:= new Platform_Msg_port(outport);
      beamings: Cone_Msg_opptr:= new Cone_Msg_port(outport);
      events: Event_Msg_opptr:= new Event_Msg_port(outport);
    end record;

    PIG_name: PDL_string_ptr:= new string'("PIG");
    procedure initialize(Z: in out PIG_type; Parent: PDL_ptr);
    procedure printPIG(Z: PIG_type);
  end PIG_pkg;
  use PIG_pkg;

  --PLATFORM DESIGNATOR FUNCTION
  function lookup_platform_designator(
                      pt: platform_designator_type;
                      enter_if_not_found: Boolean:= true)
     return platform_designator_id_type;

  --DYNAMIC TM DESIGNATOR FUNCTION
  function lookup_dyn_designator(
                    pt: PDL_string_ptr;
                    enter_if_not_found: Boolean:= true)
     return dyn_designator_id_type;

  --EXCLUDE DYN TM
  procedure exclude_dyn_module(P: PDL_ptr; name: PDL_string_ptr);

  --COLLISION PREVENTION
  procedure platforms_cant_collide(pt1, pt2: platform_designator_type);

  --ANOTHER HANDY CONSTANT
  stay_at_000: constant eqn_motion_type:=
   new eqn_motion_rec'(origin, max_PDL_duration, false, null);

  --PLATFORM DEFINER PACKAGE
  generic
    type T is private;
    type T_ptr is access T;
  package PlatformDefiner_pkg is
    procedure create_platform(
                    platform_designator: platform_designator_type;
                    name: PDL_string_ptr:= empty_PDL_string;
                    discr: PDL_string_ptr:= empty_PDL_string;
                    param: T_ptr:= null;
                    mass: float:= 0.0;
                    cross_sect_rad: float:= 1.0;
                    initial_position: point_type:= origin;
                    eqn_motion: eqn_motion_type:= stay_at_000;
                    expected_lifetime: PDL_duration_type:= max_PDL_duration;
                    birth: PDL_time_type:= Current_PDL_time);
  end PlatformDefiner_pkg;

  --PLATFORM CREATION PACKAGE
  generic
    type T is private;
    type T_ptr is access T;
    platform_designator_id: platform_designator_id_type;
```

```
  with procedure platform_creation(
                    zz: out PIG_type;
                    param: T_ptr;
                    name: PDL_string_ptr;
                    discr: PDL_string_ptr;
                    characteristic: PDL_string_ptr);
package PlatformCreator_pkg is
end PlatformCreator_pkg;

--DYNAMIC TM CREATION PACKAGE
generic
  dtm_designator_id: dyn_designator_id_type;
  with procedure dyn_creation(
                  zz1: out Cone_Msg_ipptr;
                  zz2: out Event_Msg_ipptr;
                  zz3: out Platform_Msg_ipptr;
                  platform: PDL_ptr);
package DTMCreator_pkg is
end DTMCreator_pkg;

--CONE DEFINER PACKAGE
generic
  type T is private;
  type t_ptr is access T;
package ConeDefiner_pkg is
  procedure create_cone(
              cone_designator: cone_designator_type;
              cone: cone_type:= cone_everything;
              data: T_ptr:= null;
              z: PDL_ptr);
  procedure create_cone3(
              cone_designator: cone_designator_type;
              RetAddr: Cone_RetAddr_type;
              data: T_ptr:= null;
              z: PDL_ptr);
  procedure create_cone(
              cone_designator: cone_designator_type;
              RetAddr: Cone_RetAddr_type;
              data: T_ptr:= null;
              z: PDL_ptr) renames create_cone3;
end ConeDefiner_pkg;

--USEFUL PROCEDURE CALLS TO THE SIMULATION SYSTEM.
--THESE PROCEDURES ARE RENAMED IN Procedure_pkg PACKAGE BELOW.
procedure destroy_self(z: PDL_ptr);
procedure change_my_type(
            new_designator: platform_designator_type;
            z: PDL_ptr);
function change_my_eqn_motion(
              new_eqn: eqn_motion_type;
              z: PDL_ptr)
   return eqn_motion_type;
procedure extend_my_lifetime(
              extension: PDL_duration_type:= max_PDL_duration;
              z: PDL_ptr);
procedure change_my_mass(new_mass: float:= 0.0; z: PDL_ptr);
function get_my_type(z: PDL_ptr) return platform_designator_type;
function get_my_deathtime(z: PDL_ptr) return PDL_time_type;
function get_physical_stuff(z: PDL_ptr)
   return physical_stuff_block;
function platform_position(
            time: PDL_time_type:= Current_PDL_time;
            z: PDL_ptr)
   return point_type;
function platform_position_2(
              PHY: physical_stuff_block;
```

```
                time: PDL_time_type:= Current_PDL_time)
    return point_type;
  function platform_position(
                PHY: physical_stuff_block;
                time: PDL_time_type:= Current_PDL_time)
    return point_type renames platform_position_2;
  function platform_speed(PHY: physical_stuff_block) return float;
  --FUNCTIONS FOR PLATFORM NAMES
  function platform_name(Z: PDL_ptr) return PDL_string_ptr;
  function platform_discr(Z: PDL_ptr) return PDL_string_ptr;
  function platform_type(Z: PDL_ptr) return PDL_string_ptr;
  function platform_characteristic(Z: PDL_ptr)
    return PDL_string_ptr;
end interface_procs;
use interface_procs;

--MAKE PIG_pkg VISIBLE
package PIG_pkg renames interface_procs.PIG_pkg;

--PROCEDURES PACKAGE
--RENAMES THE ROUTINES OF INTERFACE_PROCS SO THAT THE Z:PDL_ptr
--ARGUMENT RECEIVES A DEFAULT.
generic
  P: PDL_ptr;
package Procedures_pkg is
  function lookup_platform_designator(
                pt: platform_designator_type;
                enter_if_not_found: Boolean:= true)
    return platform_designator_id_type renames interface_procs.lookup_platform_designator;
  function lookup_dyn_designator(
                pt: PDL_string_ptr;
                enter_if_not_found: Boolean:= true)
    return dyn_designator_id_type renames interface_procs.lookup_dyn_designator;
  procedure exclude_dyn_module(
                P: PDL_ptr;
                name: PDL_string_ptr) renames interface_procs.exclude_dyn_module;
  procedure platforms_cant_collide(
                pt1,
                pt2: platform_designator_type) renames interface_procs.platforms_cant_collide;
  procedure destroy_self(Z: PDL_ptr:= P) renames interface_procs.destroy_self;
  procedure change_my_type(
                new_designator: platform_designator_type;
                Z: PDL_ptr:= P) renames interface_procs.change_my_type;
  function change_my_eqn_motion(
                new_eqn: eqn_motion_type;
                Z: PDL_ptr:= P)
    return eqn_motion_type renames interface_procs.change_my_eqn_motion;
  procedure extend_my_lifetime(
                extension: PDL_duration_type:= max_PDL_duration;
                Z: PDL_ptr:= P) renames interface_procs.extend_my_lifetime;
  procedure change_my_mass(
                new_mass: float:= 0.0;
                Z: PDL_ptr:= P) renames interface_procs.change_my_mass;
  function get_my_type(Z: PDL_ptr:= P)
    return platform_designator_type renames interface_procs.get_my_type;
  function get_my_deathtime(Z: PDL_ptr:= P)
    return PDL_time_type renames interface_procs.get_my_deathtime;
  function get_physical_stuff(Z: PDL_ptr:= P)
    return physical_stuff_block renames interface_procs.get_physical_stuff;
  function platform_position(
                time: PDL_time_type:= Current_PDL_time;
                Z: PDL_ptr:= P)
    return point_type renames interface_procs.platform_position;
  function platform_position(
                PHY: physical_stuff_block;
                time: PDL_time_type:= Current_PDL_time)
```

```
        return point_type renames interface_procs.platform_position_2;
      function platform_speed(PHY: physical_stuff_block)
        return float renames interface_procs.platform_speed;
      function platform_name(Z: PDL_ptr:= P)
        return PDL_string_ptr renames interface_procs.platform_name;
      function platform_discr(Z: PDL_ptr:= P)
        return PDL_string_ptr renames interface_procs.platform_discr;
      function platform_type(Z: PDL_ptr:= P)
        return PDL_string_ptr renames interface_procs.platform_type;
      function platform_characteristic(Z: PDL_ptr:= P)
        return PDL_string_ptr renames interface_procs.platform_characteristic;
    end Procedures_pkg;

    package DONT_USE is
      --NOTIFY_ANALOG_PART: THE RESULT OF CALLING THIS PROCEDURE IS
      --UNKNOWN AND (NODOUBT) DISASTEROUS. DO NOT CALL THIS PROCEDURE!
      function notify_analog_part return boolean;
      --NOTIFY_ANALOG_PART SHOULD NOT BE USED.
      --DO NOT CALL THIS PROCEDURE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    end DONT_USE;
  end Cones_n_Platforms;
```

APPENDIX E

April 1988 Specification of System_Scheduler and Example

To simulate a system of platforms orbiting through space, a program to create the initial configuration or platforms and trigger the start of the simulation is needed. This program is referred to as the *main* program. The main program establishes the initial configuration of platforms by calling the procedure *create_platform*.

The procedures *create_platform* are found in the instatiations of the *PlatformDefiner_pkg* located in each platform. As a result, the main program must "with" all of the packages representing platforms created in the initial configuration. Furthermore, the main program must "withρq the *Cones_n_Platforms* package in order to access the data types and routines needed to establish the equation of motion for the individual platforms. This package also yields access to the *PDL_pkg* package. The PDL_pkg contains the time types and the IO packages.

After the configuration is established, the main program starts the simulation by calling the procedure *start_simulation*. The main program must "with" the *System_Scheduler* package, given below:

```
with PDL_pkg;
package System_Scheduler is
  procedure start_simulation(stop_simulation_time: PDL_pkg.PDL_time_type:= 1500);
end System_Scheduler;
```

This package contains only one definition, the procedure *start_simulation*. After creating the initial configuration of platforms, the main program issues the *start_simulation* procedure call and specifies maximum amount of time the simulation will run. The time parameter is given as a *PDL_time_type*. This value can be converted to seconds, minutes, or hours using the constant *PDL_ticks_per_second*.

In this example, the program (*main_sds0*) creates two *Command_Post_Platforms*, four *Sensor_Platforms*, several *Weapons_Platforms*, and one *Russian_Missile_Base_Platform*. After creating these platform, this sample program calls the procedure *start_simulation* with a stop time of one *hour*.

Specification of the SDS0 Main Program

```
with System_Scheduler, Cones_n_Platforms, Latitude_n_Longitude,
 Sensor_Cone_Response_TM_pkg, Tracker_TM_pkg, Platform_Collision_TM_pkg,
 Russian_Missile_Base_Platform_pkg, Command_Post_pkg, Sensor_Platform_pkg,
 Weapons_Platform_pkg, Math, Vector_pkg, Debug_flags, VERDIX;

procedure main_sds0 is
  package PDL_IO renames Cones_n_Platforms.PDL_pkg.PDL_IO;
  package CnP_IP renames Cones_n_Platforms.interface_procs;
  use PDL_IO;
  use txt_io, int_io;
  use Cones_n_Platforms, System_Scheduler, Vector_pkg,
   Cones_n_Platforms.eqn_motion_pkg, Latitude_n_Longitude,
   Russian_Missile_Base_Platform_pkg, Command_Post_pkg, Sensor_Platform_pkg,
```

```
  Weapons_Platform_pkg, Math;
 use PDL_pkg;
 use Command_Post_PARAM_pkg;
 use Sensor_Platform_PARAM_pkg;
 use Weapons_Platform_PARAM_pkg;

 num_sensor_platforms: constant:= 4;

 SECONDS: constant:= PDL_ticks_per_second;
 MINUTES: constant:= 60 * SECONDS;
 HOUR: constant:= 60 * MINUTES;

 missile_base_eom: eqn_motion_type:= new_eqn_motion_rec;
 post1_eom: eqn_motion_type:= new_eqn_motion_rec;
 post1_param: Command_Post_parameterization_ptr:=
  new Command_Post_parameterization;
 post2_eom: eqn_motion_type:= new_eqn_motion_rec;
 post2_param: Command_Post_parameterization_ptr:=
  new Command_Post_parameterization;
 sensor_eom: eqn_motion_type;
 sensor_param: Sensor_Platform_parameterization_ptr;
 sensor_discr: PDL_string_ptr;
 weapon_eom: eqn_motion_type;
 weapon_param: Weapons_Platform_parameterization_ptr;
 sensor_radius: constant:= Re * 3.875;
 weapon_radius: constant:= Re * 1.125;
 weapon_discr: PDL_string_ptr;
 theta: float;
 platform_pos: vector;

begin
 put_line("The SDS0 simulation:");

 CnP_IP.platforms_cant_collide(Command_Post_designator, Command_Post_designator);
 CnP_IP.platforms_cant_collide(Sensor_Platform_designator,
  Sensor_Platform_designator);
 CnP_IP.platforms_cant_collide(Russian_Missile_Base_Platform_designator,
  Weapons_Platform_designator);

 put_line("   Creating Russian Missile Base at " &
  "(55.8 degrees E, 37.9 degrees N).");
 missile_base_eom.position:= location(37.9, 55.8);
 missile_base_eom.delta_t:= max_PDL_duration;
 missile_base_eom.back_ptr_flag:= true;
 missile_base_eom.next_rec:= missile_base_eom;
 Russian_Missile_Base_Platform_CP_pkg.create_platform(Russian_Missile_Base_Platform_designator,
  name => Russian_Missile_Base_Platform_name, initial_position => location(37.9,
  55.8), eqn_motion => missile_base_eom);
 put_line("   Creating Command Post 1 at " &
  "(255.0 degrees E, 37.5 degrees N).");
 post1_param.CP_id:= "CPost 1";
 post1_eom.position:= location(37.5, 255.0);
 post1_eom.delta_t:= max_PDL_duration;
 post1_eom.back_ptr_flag:= true;
 post1_eom.next_rec:= post1_eom;
 Command_Post_CP_pkg.create_platform(Command_Post_designator,
  name => Command_Post_name, discr => PDL_string_ptr'(
                                      new string'("1")),
                                      param => post1_param,
                                      initial_position => location(37.5,
                                      255.0),
                                      eqn_motion => post1_eom);
 put_line("   Creating Command Post 2 at " &
  "(270.0 degrees E, 37.5 degrees N).");
 post2_param.CP_id:= "CPost 2";
 post2_eom.position:= location(37.5, 270.0);
```

```
post2_eom.delta_t:= max_PDL_duration;
post2_eom.back_ptr_flag:= true;
post2_eom.next_rec:= post2_eom;
Command_Post_CP_pkg.create_platform(Command_Post_designator,
 name => Command_Post_name, discr => PDL_string_ptr'(
                                new string'("2")),
                                param => post2_param,
                                initial_position => location(37.5,
                                270.0),
                                eqn_motion => post2_eom);
put("Creating ");
put(num_sensor_platforms, 1);
put(" Sensor Platforms ");
for ij in 1 .. num_sensor_platforms loop
  sensor_eom:= new_eqn_motion_rec;
  theta:= float(ij - 1) * 2.0 * PI / float(num_sensor_platforms);
  platform_pos.x:= cos(theta) * sensor_radius;
  platform_pos.y:= sin(theta) * sensor_radius;
  platform_pos.z:= 0.0;
  sensor_eom.position:= platform_pos;
  sensor_eom.delta_t:= max_PDL_duration;
  sensor_eom.back_ptr_flag:= true;
  sensor_eom.next_rec:= sensor_eom;
  sensor_param:= new Sensor_Platform_parameterization;
  sensor_param.SP_id:= "Sensor ";
  put(sensor_param.SP_id(7 .. 7), ij);
  sensor_discr:= new string'(integer'image(ij));
  Sensor_Platform_CP_pkg.create_platform(Sensor_Platform_designator,
   name => Sensor_Platform_name, discr => sensor_discr, param => sensor_param,
   initial_position => platform_pos, eqn_motion => sensor_eom);
  put('.');
end loop;
new_line;
put("Creating ");
put(Debug_Flags.num_weapons_platforms, 1);
put(" Weapons Platforms ");
for ij in 1 .. Debug_Flags.num_weapons_platforms loop
  weapon_eom:= new_eqn_motion_rec;
  theta:= float(ij - 1) * 2.0 * PI / float(Debug_Flags.num_weapons_platforms);
  platform_pos.x:= cos(theta) * weapon_radius;
  platform_pos.y:= sin(theta) * weapon_radius;
  platform_pos.z:= 0.0;
  weapon_eom.position:= platform_pos;
  weapon_eom.delta_t:= max_PDL_duration;
  weapon_eom.back_ptr_flag:= true;
  weapon_eom.next_rec:= weapon_eom;
  weapon_param:= new Weapons_Platform_parameterization;
  weapon_param.WP_id:= "WP    ";
  put(weapon_param.WP_id(3 .. 7), ij);
  weapon_discr:= new string'(integer'image(ij));
  Weapons_Platform_CP_pkg.create_platform(Weapons_Platform_designator,
   param => weapon_param, name => Weapons_Platform_name,
   discr => weapon_discr, initial_position => platform_pos,
   eqn_motion => weapon_eom);
  put('.');
end loop;
new_line;
put_line("simulation begins.....");
start_simulation(PDL_time_type(HOUR));

end main_sds0;
```

APPENDIX F

April 1988 Specifications for RegExpOpDefiner_pkg

The *RegExpOpDefiner_pkg* package define four functions useful when specifying regular expressions of Atomic Events. These are the functions *any*, *every*, *each*, and *sequence* (defined in Chapter 4).

Specification of the RegExpOpDefiner_pkg

```
with PDL_n_PORT_pkg;
generic
  type INDEX is (<>);
  type T is array(INDEX range <>) of PDL_n_PORT_pkg.RGEXP.regular_expression;
package RegExpOpDefiner_pkg is
  use PDL_n_PORT_pkg.RGEXP;
  function any(AA: T) return regular_expression;
  function every(AA: T) return regular_expression;
  function each(AA: T) return regular_expression;
  function sequence(AA: T) return regular_expression;
end RegExpOpDefiner_pkg;
package body RegExpOpDefiner_pkg is
  function any(AA: T) return regular_expression is
  begin
    return null;
  end any;
  function every(AA: T) return regular_expression is
  begin
    return null;
  end every;
  function each(AA: T) return regular_expression is
  begin
    return null;
  end each;
  function sequence(AA: T) return regular_expression is
  begin
    return null;
  end sequence;
end RegExpOpDefiner_pkg;
```

APPENDIX G

April 1988 Specifications for the Resource Assignment Packages

The *Resource_Assignment_pkg* package defines the *dummy_invoke* package and several routines which extract information about the processes. This package also makes the package *RA* visible by renaming *PDL_n_PORT_pkg.RA*. The *RA* package is given later under the title "Internal Resource_Assignment_pkg".

The *PDL_n_PORT_pkg* package is used by the *Resource_Assignment_pkg*; however, the contents of the *PDL_n_PORT* package are **not** given in this document because it is not useful in the specification of SADMT processes. **An Ada compilation unit that uses PDL_n_PORT_pkg package is not a valid SADMT process, platform or technology module.**

Specification of the External Resource_Assignment_pkg

```
with PDL_n_Port_pkg;
package Resource_Assignment_pkg is
  package RA renames PDL_n_Port_pkg.RA;
  package EX renames RA.export;
  package process_data_extractors is
    use PDL_n_Port_pkg;
    function Node_Type(Z: PDL_ptr) return Process_Node_Type;
    function parent(Z: PDL_ptr) return PDL_ptr;
    function adam(Z: PDL_ptr) return PDL_ptr;
    function name(Z: PDL_ptr) return PDL_string_ptr;
    function discr_name(Z: PDL_ptr) return PDL_string_ptr;
    function ptype_name(Z: PDL_ptr) return PDL_string_ptr;
    function characteristics(Z: PDL_ptr) return PDL_string_ptr;
  end process_data_extractors;
  package DUMMY_INVOKE is
    type invoke_block is record
      null;
    end record;
    type invoke_ptr is access invoke_block;
    procedure compute_arrival_time(
                  L_PARAM: invoke_ptr;
                  cached_EXP: in out RA.PDExp;
                  arrival_time: out PDL_n_Port_pkg.PDL_timing.PDL_time_type;
                  this_message_length: integer:= 0);
  end DUMMY_INVOKE;
end Resource_Assignment_pkg;
```

The "Internal Resource_Assignment_pkg" is located within one of the internal (*PDL_n_PORT_pkg*) SADMT packages. (The package is located here because it affects the primative operations of SADMT.) The "Internal Resource_Assignment_pkg" is given below because it is made visible via renaming in the "External Resource_Assignment_pkg". The code listed below is extracted from the *PDL_n_PORT_pkg*, and contains the *use* and *renames* statements (after the package specification) that change its identity from *Resource_Assignment_pkg* to *RA*.

Specification of the Internal Resource_Assignment_pkg

```
package Resource_Assignment_pkg is
  package export is
    subtype PDL_magic_ptr is PDL_n_Port_pkg.PDL_magic_ptr;
  end export;
  type resource is access PDL_n_Port_pkg.resource_block;
  type priority is new integer;

  type PDExp is access PDL_n_Port_pkg.PDExp_block;
  package PDExp_routines is
    function "+" (l: PDExp) return PDExp;
    function "-" (l: PDExp) return PDExp;
    function "abs" (l: PDExp) return PDExp;
    function "+" (l, r: PDExp) return PDExp;
    function "-" (l, r: PDExp) return PDExp;
    function "*" (l, r: PDExp) return PDExp;
    function "/" (l, r: PDExp) return PDExp;
    function CONST(constant_value: float) return PDExp;
    function UNIFORM(
              low_value, high_value: float;
              seed: integer:= -1)
        return PDExp;
    function EXP(mean_ia_time: float; seed: integer:= -1)
        return PDExp;
    function NORMAL(
              mean, variance: float;
              seed: integer:= -1)
        return PDExp;
    function MESSAGE_LENGTH return PDExp;
    procedure set_seed(PDE: PDExp; seed: integer);
    procedure randomize(PDE: PDExp);
    function copy(PDE: PDExp) return PDExp;
    function select_from(PDE: PDExp) return float;
  end PDExp_routines;

  type ArrayOf_Float is array(integer range <>) of float;
  type ArrayOF_PDLstringptr is array(integer range <>) of PDL_string_ptr;

  type P_Delay_status_type is (initial, normal, timed_out, preempted,
   do_return, defer);
  type resource_list_block;
  type resource_list_ptr is access resource_list_block;
  subtype List_of_Resources is resource_list_ptr;
  type priority_list_block;
  type priority_list_ptr is access priority_list_block;
  subtype List_of_Priorities is priority_list_ptr;
  type P_Delay_state_type is record
    state: integer;
    resources: List_of_Resources;
    priorities: List_of_Priorities;
    holding_period: PDL_duration_type;
    actual_period: PDL_duration_type;
    status: P_Delay_status_type;
  end record;
  procedure Start_a_Resource_List(RL: out List_of_Resources; r: resource);
  procedure Append_to_a_Resource_List(
                      RL: in out List_of_Resources;
                      r: resource);
  procedure Start_a_Priority_List(
                    PL: out List_of_Priorities;
                    seize_priority, holding_priority: priority);
  procedure Append_to_a_Priority_List(
                      PL: out List_of_Priorities;
                      seize_priority,
                      holding_priority: priority);
```

```
  type P_Delay_parameterization_type(NumNumericParams:
   integer; NumStringParams: integer) is record
    Op: PDL_string_ptr;
    NumericParams: ArrayOf_Float(1 .. NumNumericParams);
    StringParams: ArrayOf_PDLstringptr(1 .. NumStringParams);
    Time_out_time: PDL_time_type;
  end record;

  type resource_list_block is record
    null;
  end record;
  type priority_list_block is record
    null;
  end record;
end Resource_Assignment_pkg;
use Resource_Assignment_pkg;
package RA renames Resource_Assignment_pkg;
```

APPENDIX H

April 1988 Specifications for the Resource Assignment Generic Packages

These two generic packages are used in conjuction with the resource assignment and modeling facilities of SADMT.

Specification of the ResourceModuleDefiner_pkg Package

```
with PDL_pkg, Resource_Assignment_pkg;
use Resource_Assignment_pkg;
generic
  type process_cache_block is private;
  type process_cache_ptr is access process_cache_block;
  type platform_cache_block is private;
  type platform_cache_ptr is access platform_cache_block;
  type invoke_block is private;
  type invoke_ptr is access invoke_block;
  with procedure initialization_notification(
                        Z: PDL_pkg.PDL_ptr;
                        process_cache: out process_cache_ptr;
                        platform_cache: in out platform_cache_ptr);
  with procedure transit_delay(
                P1, P2: process_cache_ptr;
                cached_EXP: in out RA.PDExp);
  with procedure compute_arrival_time(
                    L_PARAM: invoke_ptr;
                    cached_EXP: in out RA.PDExp;
                    arrival_time: out PDL_pkg.PDL_time_type;
                    this_message_length: integer:= 0);
  with procedure processing_delay_next_state(
                        P: process_cache_ptr;
                        PARAM: RA.P_Delay_parameterization_type;
                        STATE: in out RA.P_Delay_state_type);
package ResourceModuleDefiner_pkg is
  use PDL_pkg;
  procedure define_resource_allocation(P_designator: PDL_string_ptr);
end ResourceModuleDefiner_pkg;
```

Specification of the InvokeDefiner Package

```
with Resource_Assignment_pkg;
use Resource_Assignment_pkg;
generic
  type invoke_block is private;
  type invoke_ptr is access invoke_block;
package InvokeDefiner is
  function INVOKE(IBP: invoke_ptr) return RA.PDExp;
end InvokeDefiner;
```

APPENDIX I

Changing a Simple Example for Resource Assignment

The following task bodies represent the modifications which must be made to *RW_proc* and *Simple_proc* to make use of the "running example" of Chapter 5.

Revised task body for rw_proc

```
separate(RW_Process_pkg)
task body RW_Process_task is
  use timing_ops;
  Z: RW_Process_type:= null;
  buffer: Simple_Msg;
  start_up_time, last_time: PDL_time_type:= 1000;
  which_port: integer:= -50;
begin
  accept start_up(Z: RW_Process_type) do
    RW_Process_task.Z:= Z;
    make_known(Z.PDL);
  end start_up;
  declare
    package WAITING_pkg is new Wait_pkg(Z.PDL);
    use WAITING_pkg;
  begin
    wait_for_initialization;
    start_up_time:= Current_PDL_time;
    loop
      while not port_empty(Z.message_in) loop
        write_process_id(Z.PDL, "    DEQUEUING-", "|", false);
        put_msg(port_data(Z.message_in), 0);
        consume(Z.message_in);
      end loop;
      if Last_time /= Current_PDL_time then
        if (Current_PDL_time - start_up_time) mod 40 = 0 or
         (Current_PDL_time - start_up_time) mod 40 = 30 then
          buffer.time_created:= Current_PDL_time;
          buffer.last_slot:= 0;
          emit(Z.message_out, buffer);
          last_time:= Current_PDL_time;
        end if;
      end if;
      declare
        mod10: PDL_duration_type;
      begin
        mod10:= (Current_PDL_time - start_up_time) mod 10;
        if mod10 = 0 then
          wait_for_activity(Time_out => 10);
        else
          wait_for_activity(Time_out => 10 - mod10);
        end if;
      end;
    end loop;
  end;
exception
  when others =>
    write_process_full(Z.PDL, "AND THEN SOME EXCEPTION in ");
```

```
end RW_Process_task;
```

Revised task body for simple_proc

```
with PDL_pkg;
use PDL_pkg;
package Simple_process_pkg_pdconsts is
  Processing_OP: constant PDL_string_ptr:= new string'("SimpOP");
end Simple_process_pkg_pdconsts;
with Simple_process_pkg_pdconsts;
use Simple_process_pkg_pdconsts;
separate(Simple_Process_pkg)
task body Simple_Process_task is
  Z: Simple_Process_type:= null;
  buffer: Simple_msg;
begin
  accept start_up(Z: Simple_Process_type) do
    Simple_Process_task.Z:= Z;
    make_known(Z.PDL);
  end start_up;
  declare
    package WAITING_pkg is new Wait_pkg(Z.PDL);
    use WAITING_pkg;
  begin
    wait_for_initialization;
    loop
      wait_for_activity;
      buffer:= port_data(Z.message_in);
      consume(Z.message_in);
      wait(Processing_OP, NumericParams => (1 => float(Z.PRM.waittime)),
       Default_Delay => Z.PRM.waittime);
      buffer.last_slot:= buffer.last_slot + 1;
      buffer.route(buffer.last_slot):= integer(Z.PDL.process_id);
      emit(Z.message_out, buffer);
    end loop;
  end;
exception
  when others =>
    write_process_full(Z.PDL, "AND THEN SOME EXCEPTION in ", "**");
end Simple_Process_task;
```

APPENDIX J

Example of Developing a Resource Assignment Module

The following code defines a resource assignment module for the "running example" of Chapter 5. The package *alternative* defines the example procedures used to model transit delays adjusted for load. The procedure *processing_delay_next_state* defines the behavior of processing delays. Finally, the packages *RAM* (Resource Assignment Module) and *RAM_alt* are intantiated. *RAM* is instantiated without (with DUMMY) processing delays (*DI*) while *RAM_alt* is instantiated with the processing delays specified in the package *alternative*.

```
with PDL_pkg, Resource_Assignment_pkg;
use PDL_pkg;
package RA_example is
  use Resource_Assignment_pkg.RA;
  type leaflink_types is (notofinterest, rwproc, simpleproc1);
  type platform_cache_block is record
    a_machine: resource;
    resource_list: List_Of_Resources;
  end record;
  type platform_cache_ptr is access platform_cache_block;
  type ArrayOf_List_Of_Priorities is array(integer range <>) of List_Of_Priorities;
  type process_cache_block is record
    PDL: PDL_ptr;
    PLATFORM: platform_cache_ptr;
    link_type_indicator: leaflink_types:= notofinterest;
    priority_lists: ArrayOf_List_Of_Priorities(1 .. 2);
  end record;
  type process_cache_ptr is access process_cache_block;
  procedure initialization_notification(
                        Z: PDL_pkg.PDL_ptr;
                        process_cache: out process_cache_ptr;
                        platform_cache: in out platform_cache_ptr);
  procedure transit_delay(P1, P2: process_cache_ptr; cached_EXP: in out PDExp);
  package alternative is
    type invoke_block is record
      case_indicator: integer;
      earliest_arrival_time: PDL_time_type;
    end record;
    type invoke_ptr is access invoke_block;
    procedure transit_delay_2(
                  P1, P2: process_cache_ptr;
                  cached_EXP: in out PDExp);
    procedure compute_arrival_time(
                    L_PARAM: invoke_ptr;
                    cached_EXP: in out PDExp;
                    arrival_time: out PDL_pkg.PDL_time_type;
                    this_message_length: integer:= 0);
  end alternative;
  procedure processing_delay_next_state(
                        P: process_cache_ptr;
                        PARAM: P_Delay_parameterization_type;
                        STATE: in out P_Delay_state_type);
end RA_example;
package body RA_example is
  procedure initialization_notification(
                        Z: PDL_pkg PDL_ptr;
```

```
                    process_cache: out process_cache_ptr;
                    platform_cache: in out platform_cache_ptr) is separate;
  procedure transit_delay(
              P1, P2: process_cache_ptr;
              cached_EXP: in out PDExp) is separate;
  package body alternative is separate;
  procedure processing_delay_next_state(
                    P: process_cache_ptr;
                    PARAM: P_Delay_parameterization_type;
                    STATE: in out P_Delay_state_type) is separate;
end RA_example;
separate(RA_example)
package body alternative is
  procedure transit_delay_2(
              P1, P2: process_cache_ptr;
              cached_EXP: in out PDExp) is separate;
  procedure compute_arrival_time(
                L_PARAM: invoke_ptr;
                cached_EXP: in out PDExp;
                arrival_time: out PDL_pkg.PDL_time_type;
                this_message_length: integer:= 0) is separate;
end alternative;
separate(RA_example)
procedure initialization_notification(
                  Z: PDL_pkg.PDL_ptr;
                  process_cache: out process_cache_ptr;
                  platform_cache: in out platform_cache_ptr) is
  use Resource_Assignment_pkg.process_data_extractors;
begin
  if Node_Type(Z) /= leaf then
    return;
  end if;
  if platform_cache = null then
    platform_cache:= new platform_cache_block;
    Start_a_Resource_List(platform_cache.resource_list,
     platform_cache.a_machine);
  end if;
  process_cache:= new process_cache_block;
  process_cache.PDL:= Z;
  process_cache.PLATFORM:= platform_cache;
  if Name(Z).all = "RW_proc" then
    process_cache.link_type_indicator:= rwproc;
  else
    if Discr_Name(Z).all = "1" then
      process_cache.link_type_indicator:= simpleproc1;
    end if;
    if Discr_Name(Z).all = "2" then
      Start_a_Priority_List(process_cache.priority_lists(1), 10, 9);
      Start_a_Priority_List(process_cache.priority_lists(2), 6, 5);
    else
      Start_a_Priority_List(process_cache.priority_lists(1), 9, 9);
      Start_a_Priority_List(process_cache.priority_lists(2), 5, 5);
    end if;
  end if;
  return;
end initialization_notification;
separate(RA_example)
procedure transit_delay(P1, P2: process_cache_ptr; cached_EXP: in out PDExp) is
  use PDExp_routines;
begin
  case P2.link_type_indicator is
    when rwproc =>
      cached_Exp:= CONST(10.0);
    when simpleproc1 =>
      cached_Exp:= EXP(3.0);
    when notofinterest =>
```

```
      cached_Exp:= null;
  end case;
  return;
end transit_delay;
separate(RA_example)
procedure processing_delay_next_state(
                    P: process_cache_ptr;
                    PARAM: P_Delay_parameterization_type;
                    STATE: in out P_Delay_state_type) is
  use timing_ops;
begin
  if STATE.status = initial then
    STATE.state:= 0;
    STATE.resources:= P.PLATFORM.resource_list;
    STATE.status:= normal;
  end if;
  case STATE.status is
    when normal =>
      STATE.state:= STATE.state + 1;
      if STATE.state > P.priority_lists'last then
        STATE.status:= do_return;
        return;
      end if;
      STATE.priorities:= P.priority_lists(STATE.state);
      if STATE.state = 1 then
        STATE.holding_period:= PDL_duration(3.0);
      else
        STATE.holding_period:= PDL_duration(PARAM.NumericParams(1));
      end if;
    when preempted =>
      STATE.holding_period:= STATE.holding_period - STATE.actual_period;
      STATE.status:= normal;
    when others =>
      STATE.status:= do_return;
  end case;
end processing_delay_next_state;
with InvokeDefiner;
separate(RA_example.alternative)
procedure transit_delay_2(P1, P2: process_cache_ptr; cached_EXP: in out PDExp) is
  package my_INVOKE is new InvokeDefiner(invoke_block, invoke_ptr);
  use my_INVOKE;
  use PDExp_routines;
begin
  case P2.link_type_indicator is
    when rwproc =>
      cached_Exp:= INVOKE(new invoke_block'(53, 0));
    when simpleproc1 =>
      cached_Exp:= EXP(3.0);
    when notofinterest =>
      cached_Exp:= null;
  end case;
  return;
end transit_delay_2;
separate(RA_example.alternative)
procedure compute_arrival_time(
                  I_PARAM: invoke_ptr;
                  cached_EXP: in out PDExp;
                  arrival_time: out PDL_pkg.PDL_time_type;
                  this_message_length: integer:= 0) is
  FIVE_SECONDS: constant PDL_duration_type:= PDL_duration_type(5 *
   PDL_ticks_per_second);
  earliest_arrival_time: PDL_time_type
   renames I_PARAM.earliest_arrival_time;
  t: PDL_time_type;
  use timing_ops;
begin
```

```
    case I_PARAM.case_indicator is
      when 53 =>
        t:= Current_PDL_time + FIVE_SECONDS;
        if earliest_arrival_time > t then
          t:= earliest_arrival_time;
        end if;
        earliest_arrival_time:= t + FIVE_SECONDS;
        arrival_time:= t;
      when others =>
        null;
    end case;
    return;
end compute_arrival_time;
with Resource_Assignment_Pkg, ResourceModuleDefiner_pkg, RA_example;
package First_Resource_Assignment is
    use RA_example;
    package DI renames Resource_Assignment_pkg.DUMMY_INVOKE;
    package AA renames RA_example.alternative;
    package RAM is
        new ResourceModuleDefiner_pkg(
                        process_cache_block,
                        process_cache_ptr,
                        platform_cache_block,
                        platform_cache_ptr,
                        DI.invoke_block,
                        DI.invoke_ptr,
                        initialization_notification,
                        transit_delay,
                        DI.compute_arrival_time,
                        processing_delay_next_state);
    package RAM_alt is
        new ResourceModuleDefiner_pkg(
                        process_cache_block,
                        process_cache_ptr,
                        platform_cache_block,
                        platform_cache_ptr,
                        AA.invoke_block,
                        AA.invoke_ptr,
                        initialization_notification,
                        transit_delay,
                        AA.compute_arrival_time,
                        processing_delay_next_state);
end First_Resource_Assignment;
```

APPENDIX K

April 1988 Surrogate for the Process_Delay_Wait Procedure

The following "package" illustrates the behavior of the *Process_Delay_Wait* procedure.

Specification of the Process_Delay_Wait Procedure

```
with PDL_n_Port_pkg;
use PDL_n_Port_pkg;
package support_PDWAIT_surrogate is
  procedure all_or_nothing_seize(
                    STATE: in out RA.P_Delay_state_type;
                    Time_out: PDL_timing.PDL_duration_type);
end support_PDWAIT_surrogate;
with PDL_n_Port_pkg, support_PDWAIT_surrogate;
use PDL_n_Port_pkg, support_PDWAIT_surrogate;
generic
  type process_cache_block is private;
  type process_cache_ptr is access process_cache_block;
  with procedure processing_delay_next_state(
                          P: process_cache_ptr;
                          PARAM: RA.P_Delay_parameterization_type;
                          STATE: in out RA.P_Delay_state_type);
package PDWAIT_surrogate is
  procedure processing_delay_wait(
                    Op: PDL_string_ptr;
                    NumericParams: RA.ArrayOf_Float;
                    StringParams: RA.ArrayOf_PDLstringptr;
                    Default_Delay: PDL_timing.PDL_duration_type;
                    Time_out: PDL_timing.PDL_duration_type;
                    P: PDL_ptr);
end PDWAIT_surrogate;
with UNCHECKED_CONVERSION;
package body PDWAIT_surrogate is
  procedure processing_delay_wait(
                    Op: PDL_string_ptr;
                    NumericParams: RA.ArrayOf_Float;
                    StringParams: RA.ArrayOf_PDLstringptr;
                    Default_Delay: PDL_timing.PDL_duration_type;
                    Time_out: PDL_timing.PDL_duration_type;
                    P: PDL_ptr) is separate;
end PDWAIT_surrogate;
separate(PDWAIT_surrogate)
procedure processing_delay_wait(
                  Op: PDL_string_ptr;
                  NumericParams: RA.ArrayOf_Float;
                  StringParams: RA.ArrayOf_PDLstringptr;
                  Default_Delay: PDL_timing.PDL_duration_type;
                  Time_out: PDL_timing.PDL_duration_type;
                  P: PDL_ptr) is
  use PDL_timing;
  use timing_ops;
  use DSS;
  use RA;
  function cast_to_cache_ptr is
     new UNCHECKED_CONVERSION(PDL_magic_ptr,process_cache_ptr);
```

```
  PCP: process_cache_ptr;
  subtype P_Delay_param_type is RA.P_Delay_parameterization_type;
  subtype P_Delay_state_type is RA.P_Delay_state_type;
  PARAM: P_Delay_param_type(NumericParams'Last, StringParams'Last);
  STATE: P_Delay_state_type;
begin
  PCP:= cast_to_cache_ptr(P.ra_process_cache);
  PARAM:= P_Delay_param_type'(
                  NumericParams'Last,
                  StringParams'Last,
                  Op => Op,
                  NumericParams => NumericParams,
                  StringParams => StringParams,
                  Time_out_time => Current_PDL_time + Time_out);
  STATE:= P_Delay_state_type'(
                  state => 0,
                  resources => null,
                  priorities => null,
                  holding_period => 0,
                  actual_period => 0,
                  status => RA.initial);
---- in the real code there is a check here for
---- the existence of a resource assignment module
  loop
    processing_delay_next_state(PCP, PARAM, STATE);
    if STATE.status > RA.preempted then
      if STATE.status = RA.defer then
        if Time_out < Default_Delay then
          wait(Time_out, P);
        else
          wait(Default_Delay, P);
        end if;
      end if;
      return;
    end if;
    all_or_nothing_seize(STATE, PARAM.Time_out_time - Current_PDL_time);
  end loop;
end processing_delay_wait;
package body support_PDWAIT_surrogate is
  procedure all_or_nothing_seize(
                  STATE: in out RA.P_Delay_state_type;
                  Time_out: PDL_timing.PDL_duration_type) is
    use PDL_timing;
    use timing_ops;
    use DSS;
    entry_time, exit_time, good_exit_time: PDL_time_type;
    interval: PDL_duration_type;
  begin
    entry_time:= Current_PDL_time;
    if we can seize all the resources at the priorites indicated then
      if STATE.holding_period < Time_out then
        interval:= STATE.holding_period;
      else
        interval:= Time_out;
      end if;
      good_exit_time:= entry_time + interval;
      hold each resource with the priority indicated
            for the interval computed;
      exit_time:= Current_PDL_time;
      STATE.actual_period:= exit_time - entry_time;
      if exit_time < good_exit_time then
        STATE.status:= RA.preempted;
      elsif interval = STATE.holding_period then
        STATE.status:= RA.normal;
      else
        STATE.status:= RA.timed_out;
```

```
        end if;
      else
        suspend waiting either for a change in resource
            ownership or until the time-out expires;
        exit_time:= Current_PDL_time;
        STATE.actual_period:= 0;
        if (exit_time - entry_time) >= Time_out then
          STATE.status:= RA.timed_out;
        else
          STATE.status:= RA.preempted;
        end if;
      end if;
    end all_or_nothing_seize;
  end support_PDWAIT_surrogate;
```

APPENDIX L

April 1988 Message Types and Arbiter for the Army Example

The following packages define the message and port types used in the "Army" example of Chapter 2. The message and port types are listed in the *Assistants_ear_pkg* and the *Soldier_status_pkg* packages. Finally, the *Arbiter_pkg* package specification and initialization is given.

Specification of the Assistants_ear Package

```
with PortDefiner_pkg, Order_pkg, PDL_pkg;
package Assistants_ear_pkg is

  subtype Assistants_ear is Order_pkg.Order_ipptr;
  type Assistants_ear_ptr is access Assistants_ear;

  Assistants_ear_debug_class: string(1 .. 4):= "EAR-";
  procedure put_msg is
     new PDL_pkg.dummy_print_procedure(Assistants_ear);
  package PD is
     new PortDefiner_pkg(
               Assistants_ear,
               put_msg,
               "EAR",
               Assistants_ear_debug_class);

  subtype Assistants_ear_port is PD.T_port;
  subtype Assistants_ear_opptr is PD.T_opptr;
  subtype Assistants_ear_ipptr is PD.T_ipptr;

end Assistants_ear_pkg;
```

Specification of the Soldier_status Package

```
with PortDefiner_pkg, Order_pkg, PDL_pkg;
package Soldier_Status_pkg is

  type life_value is (live, dead);
  type Soldier_Status is record
    index: integer;
    health: life_value;
    ear: Order_pkg.Order_ipptr;
  end record;
  type Soldier_Status_ptr is access Soldier_Status;

  Soldier_Status_debug_class: string(1 .. 4):= "Stat";
  procedure put_msg(m: Soldier_Status; indent: integer:= 35);
  package PD is
      new PortDefiner_pkg(
                Soldier_Status,
                put_msg,
                "STATUS",
                Soldier_Status_debug_class);

  subtype Soldier_Status_port is PD.T_port;
  subtype Soldier_Status_opptr is PD.T_opptr;
  subtype Soldier_Status_ipptr is PD.T_ipptr;

  package Stat_io is
      new PDL_pkg.PDL_IO.TXT_IO.ENUMERATION_IO(life_value);
end Soldier_Status_pkg;
package body Soldier_Status_pkg is
  procedure put_msg(m: Soldier_Status; indent: integer:= 35) is
    use PDL_pkg.PDL_IO;
    use TXT_IO, INT_IO, Stat_io;
  begin
    for i in 1 .. indent loop
      put(' ');
    end loop;
    put("Index= ");
    put(m.index);
    put(", state= ");
    put(m.health);
    new_line;
  end put_msg;
end Soldier_status_pkg;
```

Specification of the Arbiter Process

```
with PDL_pkg, Soldier_Status_pkg, Assistants_ear_pkg;

package Arbiter_pkg is
  use PDL_pkg, Soldier_Status_pkg, Assistants_ear_pkg;

  package Arbiter_PARAM_pkg is
    type Arbiter_parameterization is record
      num_of_soldiers: integer:= 0;
    end record;
  end Arbiter_PARAM_pkg;
  use Arbiter_PARAM_pkg;

  type Arbiter_block;
  type Arbiter_type is access Arbiter_block;

  task type Arbiter_task is
    entry start_up(Z: Arbiter_type);
  end Arbiter_task;
  type Arbiter_task_ptr is access Arbiter_task;

  type Arbiter_block is record
    PDL: PDL_ptr:= new_PDL_block(leaf);
    SEM: Arbiter_task_ptr;
    PRM: Arbiter_parameterization;
    status_in: Soldier_Status_ipptr:= new Soldier_Status_port(inport);
    chosen_assistant: Assistants_ear_opptr:= new Assistants_ear_port(outport);
  end record;

  Arbiter_name: constant PDL_string_ptr:= new string'("Arbiter");
  Arbiter_type_name: constant PDL_string_ptr:= new string'("Arbiter");
  Arbiter_discr_name: PDL_string_ptr:= empty_string;
  Arbiter_characteristic: constant PDL_string_ptr:=
   new string'("typename=Arbiter");

  procedure initialize(
              Z: in out Arbiter_type;
              Parent: PDL_ptr;
              num_of_soldiers_param: integer:= 0;
              my_name: PDL_string_ptr:= Arbiter_name;
              discr_name: PDL_string_ptr:= Arbiter_discr_name;
              type_name: PDL_string_ptr:= Arbiter_type_name;
              characteristic: PDL_string_ptr:= Arbiter_characteristic);
end Arbiter_pkg;
package body Arbiter_pkg is
  use Soldier_Status_pkg.PD.Procedures;
  use Assistants_ear_pkg.PD.Procedures;
  use PDL_IO;
  use TXT_IO, INT_IO, TIME_IO, DURATION_IO;
  task body Arbiter_task is separate;
  procedure initialize(
              Z: in out Arbiter_type;
              Parent: PDL_ptr;
              num_of_soldiers_param: integer:= 0;
              my_name: PDL_string_ptr:= Arbiter_name;
              discr_name: PDL_string_ptr:= Arbiter_discr_name;
              type_name: PDL_string_ptr:= Arbiter_type_name;
              characteristic: PDL_string_ptr:= Arbiter_characteristic) is separate;
end Arbiter_pkg;
separate(Arbiter_pkg)
procedure initialize(
            Z: in out Arbiter_type;
            Parent: PDL_ptr;
            num_of_soldiers_param: integer:= 0;
            my_name: PDL_string_ptr:= Arbiter_name;
```

```
            discr_name: PDL_string_ptr:= Arbiter_discr_name;
            type_name: PDL_string_ptr:= Arbiter_type_name;
            characteristic: PDL_string_ptr:= Arbiter_characteristic) is
begin
  Z:= new Arbiter_block;
  declare
    status_in: Soldier_Status_ipptr renames Z.status_in;
    chosen_assistant: Assistants_ear_opptr renames Z.chosen_assistant;
    num_of_soldiers: integer renames Z.PRM.num_of_soldiers;
    MYSELF: PDL_ptr renames Z.PDL;
  begin
    set_process_parent(Z.PDL, Parent, my_name, discr_name, characteristic);
    if init_debug_level > 130 then
      write_process_full(MYSELF, "*init> ", " before start_up");
    end if;
    Z.PRM.num_of_soldiers:= num_of_soldiers_param;
    initialize(Z.status_in, Z.PDL, "portname=status_in");
    initialize(Z.chosen_assistant, Z.PDL, "portname=chosen_assistant");
  end;
  Z.SEM:= new Arbiter_task;
  Z.SEM.start_up(Z);

  if init_debug_level > 130 then
    write_process_full(Z.PDL, "*init> ", " after start_up");
  end if;
exception
  when others =>
    write_process_full(Z.PDL, "***Some error in ", "**");
end initialize;
```

References

[Bell and Newell 71]
Bell, C. Gordon and Allen Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[Bruno 84]
Bruno, G. "Using Ada for Discrete Event Simulationpq, *Software - Practice and Experience*, Vol 14(7), pp. 685-695 (July 1984).

[WIS 86]
Byrnes, Christopher and Richard Hilliard II, "The WIS Ada Design Language", Draft, May 1986.

[LRM 83]
"The Ada Programming Language Reference Manual", ANSI/MILSTD 1815A, US Dept. of Defense, US Government Printing Office, 1983.

[Luckham 87]
Luckham, D. C., Helmbold, D. P., Menldal, S., Bryan, D. L., and Haberler, M. A., "Task Sequencing Language for Specifying Distributed Ada Systems – TSL-1", Stanford University, 1987.

[Luckham and Henke 85]
Luckham, David C. and Friedrich W. Henke, "An Overview of Anna, A Specification Language for Ada", *IEEE Software*, March 1985, pp. 9-22.

[SofTech 84]
"JAMPS PDL Guide", submitted to U.S. Air Force Systems Command, Aeronautical Systems Division, ADOL, prepared by SofTech, Inc., October, 1984.

[Kappel 87]
Kappel, Michael R., Cathy Jo Linn, and Joseph L. Linn, "SAGEN User's Guide", IDA, Draft paper, April 1988.

[Linn 87]
Linn, Cathy Jo, Joseph L. Linn, Michael R. Kappel, and John Salasin, "Strategic Defense Initiative Ada Process Description Language", Version 1.00, IDA Paper P-1983, Draft paper, August 1987.

[Lipton 87]
Lipton, Richard, (Princeton University), Presentation at the ISI Simulation Workshop, April 30, 1987.

[Cohen 87]
Cohen, Howard, Stephen Edwards, Joseph L. Linn, Cathy Jo Linn, Cy D. Ardoin, "A Simple Example of an SADMT Architecture Specification", IDA Paper P-2036, Draft Paper, April 1988.

Distribution List for P-2035

Sponsor

LTC Jon Rindt — 1 copy
SDIO
Pentagon
BM/C3
1E149
Washington, DC 20301-7100

CAPT David Hart — 1 copy
SDIO
Pentagon
BM/C3
1E149
Washington, DC 20301-7100

LTC Chuck Lillie — 1 copy
SDIO
Pentagon
BM/C3
1E149
Washington, DC 20301-7100

LTC Pete Sowa — 1 copy
SDIO
Pentagon
BM/C3
1E149
Washington, DC 20301-7100

Other

Defense Technical Information Center — 2 copies
Cameron Station
Alexandria, VA 22314

(Each should receive 1 copy.)

F.R. Albrow
MoD (PE) RSRE
St. Andrews Road
Great Malvern
Worcester WR14 3PS
England

John Anderson
11569 Hicks Court
Manassas, VA 22111

Jim Armitage
GTE SSD
1 Research Dr.
Westborough, MA 01581

David Audley, Associate
Financial Strategies
Prudential Bache Securities
26th Floor
199 Water
New York, NY 10292

Dr. Algirdas Avizienis
Computer Science Department
4731 Boelter Hall
University of California, Los Angeles
Los Angeles, CA 90024

Dan Baker
TASC
55 Walkers Brook Drive
Redding, MA 01867

Gary H. Barber
Program Manager
Intermetrics, Inc.
1100 Hercules, Suite 300
Houston, TX 77058

John Barry
SJ-72
Rockwell
P.O. Box 3644
Seal Beach, CA 90740-7644

Elizabeth Bently
Marketing Coordinator
Research Triangle Institute
P.O. Box 12184
Research Triangle Park, N.C. 27709

Cheryl Bittner
General Electric
Box 8555, Bldg. 19, Suite 200
Philadelphia, PA 19101

Grady Booch
Director, Software Engineering Program
Rational
1501 Salado Dr.
Mountain View, CA 94043

Gina Bowden
MS 202
Teledyne Brown Engineering
Cummings Research Park
Huntsville, AL 35807

James M. Boyle
Mathematics & Computer Science Division
Argonne National Laboratory
Argonne, IL 60549-4844

Craig Bredin
GTE Strategic Systems Division
3322 South Memorial Parkway
Suite 53
P.O. Box 14009
Huntsville, AL 35185

Capt. John R. Brill
USAF Space Division/ALR
DET 3 AFALC
Box 92960 WPC
Los Angeles, CA 90009

Alton L. Brintzenhoff
Manager, Ada Technology Operating Center
Syscon Corporation
3990 Sherman St.
San Diego, CA 92110

Dr. James C. Browne
Computation Center
Department of Computer Science
University of Texas at Austin
Austin, TX 78712

Dr. Robert R. Brown
Rand Corp.
1700 Main St.
P.O. Box 2138
Santa Monida, CA 90406

Miguel Carrio
Teledyne Brown Engineering
3700 Pender Dr.
Fairfax, VA 22030

Tom Cashion
CALSPAN Corporation
P.O. Box 9X
Lexington, MA 02123

Virginia Castor, Director
Ada Joint Program Office
1211 Fern St., Room C-107
Arlington, VA 22202

Yvonne Cekel
Marketing Services Manager
Cadre Technologies, Inc.
222 Richmond St.
Providence, RI 02903

Bijoy G. Chatterjee
Deputy Director
Advanced Technology Systems
ESL/TRW
495 Java Dr.
P.O. Box 3510
Sunnyvale, CA 3510

John L. Chinn
Manager, Advanced Technology program
General Electric Space Systems Division
4041 North First St.
San Jose, CA 95134

Starla Christakos
AFATL/SAI
Eglin Air Force Base, FL 320542

Dr. Joe Clema
ECAC/IITRI
185 Admiral Cochrane Drive
Annapolis, MD 21401

Karen Coates
Program Manager, Advanced Technology
ESL/TRW
495 Java Dr.
P.O. Box 3510
Sunnyvale, CA 94088-3510

Susan Coatney
Information Science Institute
University of Southern California
4676 Admiralty Way
Marina del Ray, CA 90292-5950

Mr. Danny Cohen
Director, System Division
Information Science Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292-6695

Christopher F. Cole
Marketing Representative
Technology Programs
General Electric Company
Space Systems Division
Valley Forge Space Center
P.O. Box 8555
Philadelphia, PA 19101

Carol Combs
National Security Agency
9800 Savage Road
Ft. Meade, MD 20755-6000

Edward R. Comer
Software Productivity Solutions, Inc.
122 North 4th Av.
Indialantic, FL 32903

Lawrence L. Cone
Cone Software Laboratory
312 East Summit Av.
Haddonfield, N.J. 08033

Robert P. Cook
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22903

Chuck Cooper
Control Data Corporation
901 E. 78th Street
MS BMW 03M
Minneapolis, MN 55420

Lee Cooper
Advanced Technology
2121 Crystal Drive, Suite 200
Arlington, VA 22202

Mark Cosby
Science Applications International Corp.
1710 Goodridge Drive
McLean, VA 22012

L. Cristina
USASD
ATTN: DASD-H-SBY
P.O. 1500
106 Wynn Dr.
Huntsville, AL 35807-3801

Vincent Dambrauskas
Technical Director
Washington Technical Center
Strategic Systems Division
GTE Government Systems Corporation
6850 Versar Center, Suite 354
Springfield, VA 22151-4196

Samuel A. DeNitto
Romse Air Development Center
RADC/COE, Bldg. 3
Griffis AFB, NY 13441-5700

Cameron M.G. Donaldson
Software Productivity Solutions, Inc.
122 North 4th Av.
Indialantic, FL 32903

Ralph Duncan
Control Data Government Systems
300 Embassy Row
Atlanta, GA 30328

Stephen Edwards
1-55 Caltech
Pasadena, CA 91126

Jim Egolf
Ford Aerospace & Computer Corp.
10440 State Hwy. 83
Colorado Springs, CO 80908

David A. Fisher
Incremental Systems Corporation
319 S. Craig St.
Pittsburgh, PA 15213

Dave Fittz
STARS Program Office
OUS
OUSDRE (R&AT/CET)
3D139
1211 Fern St., C-112
Washington, D.C. 20301-3081

Michel A. Floyd
Integrated Systems Inc.
101 University Av.
Palo Alto, CA 94301-1695

Richard Frase
SRS Technologies
1500 Wilson Blvd., Suite 800
P.O. Box 12707
Arlington, VA 22209-8707

George Gearn
Applied Researsch & Engineering
7 Railroad Avenue, Suite F
Bedford, MA 01730

Victor Giddings
MITRE Corporation
Burlington Road
Bedford, MA 01730

Claren Giese
SDIO
Pentagon
T/KE
Washington, DC 20301-7100

Colin Gilyeat
Advanced Technology
2121 Crystal Drive
Arlington, VA 22202

Robert T. Goettge
Advanced Systems Technology
12200 East Briarwood Av.
Suite 260
Englewood, CO 80112

H.T. Goranson
Sirius Inc.
P.O. Box 9258
760 Lynnhaven Parkway
Virginia Beach, VA 23452

Barbara Guyette
Marketing Specialist
Ada Products Division
Intermetrics, Inc.
733 Concord Av.
Cambridge, MA 02138

Sarah Hadley
National Security Agency
9800 Savage Road
Fort Meade, MD 20755-6000

Shmuel Halevi
Ad Cad Inc.
University Place, Suite 200
124 Mt. Auburn St.
Cambridge, MA 02138

Robert Haley
Director, SDI Programs
Cray Research, Inc.
1331 Pennsylvania Avenue, N.W.
Suite 1331 North
Washington, DC 2004

Margaret Hamilton, President
Hamilton Technologies, Inc.
17 Inman St.
Cambridge, MA 02139

Duane Harder
MS B-218
Los Alamos National Laboratory
Los Alamos, NM 87545

Evans C. Harrigan
Software Consultant
Cray Research, Inc.
2130 Main Street., Suite 280
Huntington Beach, CA 92648

Hal Hart
TRW Defense Systems Group
One Space Park
Redondo Beach, CA 90278

Goran Hemdahl
Technical Director
Advanced Systems Architectures
Johnson House, 73-79 Park Street
Camberley, Surrey GU15 3PE United Kingdom

Dale B. Henderson
Los Alamos National Laboratory
Receiving Department
Bldg. SM-30
Bikini Road
Los Alamos, NM 87545

John W. Hendricks
Systems Technologies, Inc.
242 Ocean Drive West
Stamford, CT 06902

Stephan L. Hise
Advanced Development Programs
Marketing Department - MS 1112
Westinghouse Electric Corporation
Defense Group
Friendship Site
Box 1693
Baltimore, MD 21203

Jung Pyo Hong
Los Alamos National Lab
MS K488
P.O. Box 1633
Los Alamos, NM 87544

Don Horne
SRS Technologies
1500 Wilson Blvd., Suite 800
Arlington, VA 22209-8707

Bill Horton
MITRE Corp.
1259 Lake Plaza Drive
Colorado Springs, CO 80906

Greg Janee
General Research Corp.
5383 Hollister Avenue
Santa Barbara, CA 93111

Andy Jazwinski, Director
Advanced Development
TASC - The Analytic Sciences Corporation
1700 N. Moore St., Suite 1220
Arlington, VA 22209

James R. Jill
Manager, Advanced Technologies
NTB Design
Martin Marietta Information & Communications Systems
P.O. Box 1260
Denver, CO 80201-1260

Sumalee Johnson
Rockwell Institute
P.O. Box 3644
Seal Beach, CA 90704-7644

W.G. (Gray) Jones
Science Applications International Corporation
1710 Goodridge Drive
McLean, VA 22102

Irene G. Kazakova
Director, Marketing
Interactive Development Environments
150 Fourth Street, Suite 210
San Francisco, CA 94103

Peter Keenan
Science Ltd.
Wavendon Towe
Milton, Keynes
England MK17-8LX

Judy Kerner
TRW R2/1134
One Space Park
Redondo Beach, CA 90278

Rebecca Kidd
General Research Corp.
307 Wynn Drive
Huntsville, AL 35805

Virginia P. Kobler
Chief, Technology Branch
Battle Management Division
Department of the Army
Office of the Chief of Staff
U.S. Army Strategic Defense Command
P.O. Box 1500
Huntsville, AL 35807-3801

Dr. Ijur Kulikov
Intermetrics
607 Louis Drive
Warminster, PA 18974

Lt. Ann Kuo
ESD/ATS
Hanscom AFB, MA 07831

John Michael Lake
2311 Galen Dr. #7
Champaign, IL 61821

John Latimer
Teledyne Brown Engineering
300 Sparkman Dr.
MS 44
Hunstville, AL 35807

Steve Layton
Senior Software Engineer
Martin Marietta Denver Aerospace
MS L0425
P.O. Box 179
Denver, CO 80201

Larry L. Lehman
Integrated Systems Inc.
2500 Mission College Road
Santa Clara, CA 95054

Eric Leighninger
Dynamics Research
60 Frontage Road
Andover, MA 01810

Peter Lempp
Software Products and Services, Inc.
14 East 38th Street, 14th Floor
New York, NY 10016

Bob Liley
Rockwell International Corporation
2600 West Minister Blvd.
Seal Beach, CA 90740-7644

Frank Poslajko
U.S. Army SDC
CSSD-H-SI
Huntsville, AL 35807-3801

Brian Smith
Mathematics & Computer Science Div.
Argonne National Laboratory
Building 221, Room C-219
9700 South Cass Avenue
Argonne, IL 60439-4844

Norman G. Snyder
Director of Software Services
Jodgrey Associates, Inc.
462 Highfield Ct.
Severna Park, MD 21146

J.R. Southern
USA-SDC
DASD-H-SBD
106 Wynn Drive
Huntsville, AL 35807-3801

Henry Sowizral
Schlumberger Palo Alto Research
3340 Hillview Avenue
Palo Alto, CA 94304

Stephen L. Squires
DARPA
Information Processing Techniques Office
1400 Wilson Blvd.
Arlington, VA 22209

C.E.R. Story
EASAMS Ltd.
Lyon Way, Frimley Road
Camberley, Surrey GU16 5EX

Dr. Richard D. Stutzke
Science Applications International Corporation
1710 Goodridge Dr.
McLean, VA 22102

Agapi Svolou
Senior Scientist
Manager of Software Science
Mellon Institute
Computer Engineering Center
4616 Henry St.
Pittsburgh, PA 15213-2683

Kathy Tammen
General Research Corp.
P.O. Box 6770
Santa Barbara, CA 93160-6770

Kenneth C. Taormina
Director, Anlaysis and Technology Requirements
Teledyne Brown Engineering
West Oaks Executive Park
3700 Pender Dr.
Fairfax, VA 22030

Edward Town
Rockwell International Corp.
2600 West Minister Blvd.
Seal Beach, CA 90740-7644

Larry Tubbs
US Army Strategic Defense Command
DASH-H-5B
106 Wynn Dr.
Huntsville, AL 35807

Charles M. Vairin
Martin Marietta Denver Aerospace
MS L8079
P.O. Box 179
700 W. Mineral Avenue
Littleton, CO 80201

Dr. Brooks Van Horne
TRW
One Federal Systems Park Drive
Fairfax, VA 22033

John L. Walsh
Riverside Research Institute
Washington Research Office
1701 North Fort Myer Drive, Suite 700
Arlington, VA 22209

G. Karl Warmbrod
SPARTA, Inc.
7926 Jones Branch Road
Suite 1070
Mclean, VA 22180

Erwin H. Warshawsky, President
JRS Research Laboratories, Inc.
202 W. Lincoln Av.
Orange, CA 92665-1040

Capt. Charles R. Waryk
OSD/SDIO/S/SA
Pentagon
Room 1E149
Washington, DC 20301-7100

Anthony Wasserman, President
Interactive Development Environments
150 Fourth St., Suite 210
San Francisco, CA 94103

Gio C. Weiderhold
Department of Computer Science
Stanford University
Stanford, CA 94305-2085

Bruce White
500 Montezuma Avenue, Suite 118
Santa Fe, NM 87501

Dave J. Whitley
Software Engineering Section
SCICON, Ltd.
Wavedon Tower
Wavedon Village
Milton Keynes, England MK178LX

John Wiley
BDM Corporation
2227 Drake Avenue
Huntsville, AL 83505

John D. Wolfe
Programmer/Analyst
Software Consulting Specialists, Inc.
P.O. Box 15367
Fort Wayne, IN 46885

Juan A. Wood
Los Alamos National Laboratory
Receiving Department
Bldg. SM-30
Bikini Road
Los Alamos, NM 87545

Richard M. Wright
0/96-01 B/30E
2100 East St. Elmo Road
Austin, TX 78744

Robert C. Yost
Corporate Vice-President
Director, Defense Research & Analysis Operation
SAIC (Science Applications International Corporation)
1710 Goodridge Dr.
McLean, VA 22102

Christine Youngblut
17021 Sioux Lane
Gaithersburg, MD 20878

Steve Zelazny
Science Applications International Corporation
4232 Ridge Lea Road
Amherst, NY 14226

Gerald A. Zionic
NTB Program Director
Martin Marietta Information & Communication Systems
P.O. Box 1260
Denver, CO 80201-1260

CSED Review Panel

Dr. Dan Alpert, Director — 1 copy
Center for Advanced Study
University of Illinois
912 W. Illinois Street
Urbana, Illinois 61801

Dr. Barry W. Boehm — 1 copy
TRW Defense Systems Group
MS 2-2304
One Space Park
Redondo Beach, CA 90278

Dr. Ruth Davis — 1 copy
The Pymatuning Group, Inc.
2000 N. 15th Street, Suite 707
Arlington, VA 22201

Dr. Larry E. Druffel — 1 copy
Software Engineering Institute
Shadyside Place
480 South Aiken Av.
Pittsburgh, PA 15231

Dr. C.E. Hutchinson, Dean — 1 copy
Thayer School of Engineering
Dartmouth College
Hanover, NH 03755

Mr. A.J. Jordano — 1 copy
Manager, Systems & Software
Engineering Headquarters
Federal Systems Division
6600 Rockledge Dr.
Bethesda, MD 20817

Mr. Robert K. Lehto — 1 copy
Mainstay
302 Mill St.
Occoquan, VA 22125

Mr. Oliver Selfridge — 1 copy
45 Percy Road
Lexington, MA 02173

IDA

General W.Y. Smith, HQ	1 copy
Mr. Seymour Deitchman, HQ	1 copy
Mr. Philip Major, HQ	1 copy
Ms. Karen H. Weber, HQ	1 copy
Dr. Jack Kramer, CSED	1 copy
Dr. Robert I. Winner, CSED	1 copy
Dr. John Salasin, CSED	1 copy
Dr. Cathy J. Linn, CSED	100 copies
Dr. Joseph L. Linn, CSED	1 copy
Mr. Michael R. Kappel, CSED	1 copy
Mr. Howard Cohen, CSED	1 copy
Ms. Deborah Heystek, CSED	1 copy
Dr. Reginald Meeson, CSED	1 copy
Dr. Karen Gordon, CSED	1 copy
Dr. Norman Howes, CSED	1 copy
Dr. Dennis Fife, CSED	1 copy
Dr. Cy Ardoin, CSED	1 copy
Ms. Julia Sensiba, CSED	2 copies
Albert J. Perrella, Jr., STD	1 copy
IDA Control & Distribution Vault	3 copies

END

DATED

FILM

8-88

DTIC